INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

THE TOWERS OF HANOI PUZZLE

AND

THE PADLUC CRYPTOSYSTEM

By

Susan Azimi-Gass

A dissertation submitted to the Graduate Faculty in Computer Science
in partial fulfillment of the requirements for the degree of Doctor of Philosophy,
The City University of New York

1998

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: 9830682

Copyright 1998 by
Azimi-Gass, Susan

All rights reserved.

UMI Microform 9830682
Copyright 1998, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

© 1998

Susan Azimi-Gass

All Rights Reserved

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This manuscript has been read and accepted
for the Graduate Faculty in Computer Science
in satisfaction of the dissertation requirement
for the degree of Doctor of Philosophy

Feb 2, 1998
Date

Michael Anshel
Professor Michael Anshel
Chair of Examining Committee

Feb. 3, 1998
Date

Stanley Habib
Professor Stanley Habib
Executive Officer

Professor Stefan Burr

Professor Christina Zamfirescu

Assistant Professor Flora Keshishian

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTRACT

THE TOWERS OF HANOI PUZZLE

AND

THE PADLUC CRYPTOSYSTEM

By

Susan Azimi-Gass

Advisor: Professor Michael Anshel

The Padluc algorithm, presented and tested, encrypts plaintext through the use of a one-time pad generated by breaking into the exponentially long sequence of moves it finds in solving Edouard Lucas' Towers of Hanoi Puzzle.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGEMENTS

It is with abiding affection that I express my gratitude to my mentor, Professor Michael Anshel, who set my course and to his wife Ziva and daughters Iris and Daphne, who puffed my sails in the doldrums; to Professor Stefan Burr, whose rigorous critique spurred much improvement of the manuscript; to Professor Stanley Habib, whose high standards have been an aspiration; to Professor Christina Zamfirescu, for her intellectual challenge and emotional support; to Dr. Flora Keshishian, upon whose patience and wisdom I drew heavily and who helped me always to keep my eye on the ball; to Professor Steven Minsker, who so graciously and painstakingly responded to an inquiry out of the blue; to Sam Nelson, who generously extended his prodigious programming expertise; and to my remarkable friends Holly Gregory and her family, Dr. Firouzeh Kashani-Sabet, Dr. and Mrs. Mohammad Kashani-Sabet, Doris Bodine, Natalie Hammerman and Lida Khorsandi, the mettle of whose friendship was proved in the crucible of this project. Lovingly do I acknowledge my son Darius, who, since his birth, has been the greatest example of endurance and encouragement; my husband Tom, who has been true to his vow to take me "for better or for worse;" my Mother, Touran, to whom I owe a debt without measure; my sisters Nasrin, Nahid and Winifred and their families with whom I proudly share my accomplishments; and Manus and Estella Gass, the Mom and Dad who so happily came to me by way of marriage, for holding me up and easing this passage.

It has taken a village, this but a sampling of the census.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

Chapter 1 1

1.1 Introduction 1
1.2 Historical Perspective 2
1.3 The Towers of Hanoi 5
1.4 Solutions and Extensions 8

Chapter 2 12

2.1 Definitions and Terminology 12
2.2 The Vigenere Cipher 15
2.3 One-Time Pad 16
2.4 Shift Registers and Finite State Machines 18

2.4.1 Introduction 18
2.4.2 Finite State Machines 18
2.4.3 Shift Registers 21
2.4.4 Linear Feedback Shift Registers 21

2.5 Stream Ciphers 24

Chapter 3 26

3.1 Quadradic Residues 27
3.2 Huffman Codes 28
3.3 Chi-Square Test 30
3.4 Kasiski Examination 32
3.5 Procedure getSequence 32
3.6 Padluc Algorithm Layout 34
3.7 Padluc Example 35

Chapter 4 38

4.1 Conclusion 38
4.2 Recommendations for Further Study 39

4.2.1 Towers of Hanoi, Huffman Code and Authentication 39
4.2.2 The Towers of Hanoi Automaton and Encryption Application 44
4.2.3 The Super Towers of Hanoi (Superhanoi) -- Digital Signature 46
4.2.4 The Hanoi Graph, Fractals and Password Security 47
4.2.5 Morphisms and the TOH Squarefree Sequences -- Encryption Application 50

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix 54

A.1 The Padluc Algorithm -- C Implementation 54
A.2 Sample Plaintext 73
A.3 Sample Output of the Padluc Implementation 76
A.4 Chi-Square Test Result 82
A.5 Kasiski Test Result 87
A.6 The Padluc Algorithm and Three Variations 87
A.7 The Padluc Variations -- C Implementations 90
A.8 Chi-Square and Kasiski Test Results for
Four Alternative Padluc Algorithms 113

Bibliography 116

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF FIGURES

Figure 1.1 Towers of Hanoi -- The Initial Configuration for 8 Disks 5

Figure 1.2 The Final Destination for 8 Disks 6

Figure 1.3(a) TOH -- Initial Configuration for *n* Disks 8

Figure 1.3(b) Transfer of the Top *n*-1 Disks 8

Figure 1.3(c) Transfer of the *n*th Disk to the Target Peg 9

Figure 1.3(d) Transfer of *n*-1 Disks to the Target Peg 9

Figure 2.1 A Cryptosystem 13

Figure 2.2 Function Table of a Finite State Machine 20

Figure 2.3 A State Diagram for a Finite State Machine 20

Figure 2.4 A Shift Register 21

Figure 2.5 A Feedback Shift Register 22

Figure 2.6 A 3-bit Linear Feedback Shift Register 22

Figure 2.7 A Stream Cipher 24

Figure 4.1 Coded 3-Disk Towers of Hanoi 41

Figures 4.2(a), 4.2(b) Huffman Trees 42

Figure 4.3 2-Automaton for Finding the *n*th TOH Move 44

Figure 4.4 The Initial Distribution of Disks in a Superhanoi Puzzle 46

Figure 4.5 The Destination Tower 46

Figure 4.6 The Hanoi Graph H_3 48

Figure 4.7 The Sierpinski Gasket 49

Figure 4.8 The Hanoi Graph H_5 for 5 Disks 49

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

PREFACE

We are about to embark upon an exploration of a puzzle which has been the subject of myth and fancy. It was popularized by one of the world's most accomplished recreational mathematicians, who, despite being unknown for cryptological writing, made teasing reference to crytography in the introduction to his book on number theory. We will demonstrate the latent power of this parlor game in generating one-time pads for use in encryption.

The discussion will follow a natural progress from history and definitions through application:

Chapter 1. Introduction to Édouard A. Lucas and the line of his pursuits thus far followed by cryptographers; discussion of the Towers of Hanoi puzzle.

Chapter 2. Definitions and descriptions of terms and concepts needed to develop a system of one-time pad cryptography.

Chapter 3. Proposal of the Padluc encryption algorithm utilizing the Towers of Hanoi puzzle to provide the lock and key to encrypt plaintext messages

Chapter 4. Conclusion and Recommendations for Further Study

Discussion of Huffman Codes and ambiguity. *Proposal* for an authentication scheme using the ambiguity due to Huffman coding in the Towers of Hanoi sequence.

Discussion of a 2-automaton for deriving the *n*th move of the Towers of Hanoi puzzle. *Proposal* for an encryption system based upon this automaton.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Discussion of the Superhanoi variation. *Proposal* for a digital signature scheme employing this variation of the puzzle.

Discussion of the Hanoi Graph and fractal graph. *Proposal* for employing the fractal generated by the TOH in a password security scheme.

Discussion of morphisms and the squarefree sequences of the puzzle. *Proposal* of an encryption algorithm based upon these squarefree sequences.

Appendix Implementation of the Padluc algorithm and three of its variations using the C programming language. Subjecting the Padluc algorithm to Chi-square and Kasiski testing. Reporting of test results.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 1

1.1 Introduction

The development and distribution of micro-electronics in the last quarter of the 20th century has brought a vast number of people easy access to advanced computer systems. The irresistible force of computer hacking, industrial spying and government monitoring of confidential data has propelled a counter interest in locking computer data behind immovable security walls -- accelerating the momentum of research and development in the field of cryptography. Technology once exclusive to military security now serves ordinary citizens; yet worldwide government restrictions on public encryption have allowed only “toy ciphers” to be built.

Schneier’s observation [46] that the NSA has “finally allowed” a toy algorithm with a 160-bit repeated key, for voice privacy in digital cell phones, has been a motivation for this dissertation. The toy we have chosen to focus on is generated by a puzzle introduced by the French number theorist François Édouard Anatole Lucas (1842-1891) [26]. Lucas’ puzzles have become diversely used tools of research. His most noteworthy contribution is known as the Towers of Hanoi puzzle.

Lucas’ research in number theory has contributed to and influenced current cryptographic systems. Surprisingly, his pioneering exploration of puzzles and related topics have not been extensively applied to the field of cryptography. It is our plan to remedy this.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.2 Historical Perspective

From passing biographical notes in the mathematical literature, one gets a sense of Lucas as an exuberant, entertaining and spontaneous high school teacher chafing under syllabuses; a self-propelled number theorist set upon independent investigation. He is referred to as a recreational mathematician as his discoveries have preceded their application.

With a keen appreciation of mathematical history and novelty, Lucas seems to have been cast in the role of netting a numberless populace in the enchantment of numbers. Lucas promoted the popularity of mathematics by amassing (and donating to a Paris museum) a collection of calculating machines and by publishing an original cycle of scientific puzzles which won a gold medal at the 1889 World's Fair. These puzzles have been lost and so remain an open dissertation topic. He worked as a member of the French Commission to bring out the works of Fermat [24] and manually calculated the largest prime of his day [28].

Lucas was graduated from the highly regarded École Normale in 1864 and was employed at the Paris Observatory which he rejoined after military service in the Franco-Prussian War in 1870-71. From 1872 to his death at the age of 49, Edouard Lucas taught higher mathematics at three high schools [28].

In Lucas' observation that he lived "in a time and in a country where higher mathematics is forsaken by mathematicians and public education," [28] he has made himself a contemporary and countryman of everyone.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It is remarkable that so much of Lucas' work, although seeming to stand quite apart from cryptology, is the focus of present day cryptologists, who use his sequences and primality tests and it is to be hoped, hereafter, his puzzles.

Large primes are now prized in the field of cryptography [47]. Long before this was so, the discovery of the largest prime had brought great honor to many a mathematician. For centuries preceding 1950, the calculation of primes was accomplished by means of pen and paper, and before that, stick and papyrus. Euclid includes primes in his discussion of perfect numbers (numbers which are equal to the sum of their divisors, for example: 6 = 1+2+3) [40].

The Mersenne primes, named for the French friar Martin Mersenne (1588-1648) who performed a vast amount of manual calculation on perfect numbers, appear in early mathematical literature [40]. Mersenne primes take the form:

$$M_p = 2^p - 1,$$

where p is itself a prime. For example:

$$M_3 = 2^3 - 1 = 7.$$

This is not true in all cases; there are values for p where M is not a prime:

$$M_{11} = 2^{11} - 1 = 2047 = 23 \cdot 89.$$

Therefore the Mersenne numbers have to be tested for primality.

In 1750 the Swiss mathematician Euler presented the 8th Mersenne prime, for p = 31, the largest prime known for a century [40]. Euler went on to state conditions for Mersenne primes to be composite numbers for certain values of p; this was later proved by

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Lucas, when in 1876 he found a method for verifying Mersenne primes. Lucas used his method to discover the primality of the following 39 digit number:

$$M_{127} = 170141183460469231731687303715884105727$$

a fact verified by E. Fauquembergue in 1914 [48]. Lucas' prime remained the largest on record until a discovery in 1951 by Ferrier, using a desk calculating machine. Miller and Wheeler, in England, then discovered several more primes, the last of which was surpassed by p = 2281, discovered in 1952 by Lehmer and Robinson, who utilized the SWAC computer in California [23]. Thirty Mersenne primes were verified by 1985, M_{216091} being the last [48].

The Lucas test of 1876, simplified in 1930 by the Lucas-Lehmer test [40] has enjoyed continued use. The conjecture that $M_{13} = 2^{8191} - 1$ is prime was disproved by the Lucas-Lehmer theorem in a hundred hour computer run. Adelman, a developer of RSA -- an achievement in public key cryptography, is seeking large primes for cryptosystems. In 1980, Adelman and Rumely constructed a test, which, with refinements by Cohen and Lenstra, allows a one-hundred digit number to be tested for primality in one minute and a one-thousand digit number in about a week [47].

In addition to his primality testing, Lucas' study of sequences is also serving the cause of encryption. Sequences of numbers in which every succeeding number is the sum of the two preceding numbers (with initial conditions $L_1 = 1$ and $L_2 = 3$ [23, 50] or $L_1 = 2$ and $L_2 = 1$ [47]) have come to be known as *Lucas numbers*: 1, 3, 4, 7, 11, 18, 29, 47 . . . .

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Lucas numbers start with different initial conditions from Fibonacci numbers (whose initial conditions of $F_1 = F_2 = 1$ generate: 1, 1, 2, 3, 5, 8, ,13, 21, . . .) but obey the same recursion [47]. This sequence has been used in the design of LUC, a public key system [51].

The present work appears to be the first cryptological use made of a Lucas puzzle. Yet, in so doing, we might only be deciphering Édouard Lucas' encrypted intention. He, himself, referred to encoded diplomatic communications in an aside to the preface of his number theory treatise [35]. He scrambled his name into an anagram and left obscure clues as to the binary nature of the Towers of Hanoi puzzle. It is tantalizing to think that the view from the Towers may reveal Lucas' own vision of an encryption technique and other aspects of security.

1.3 The Towers of Hanoi

In 1883, the Towers of Hanoi puzzle was introduced in the mathematical literature by "Professor Claus" more fully, N. Clause de Siam [43]. This puzzle consisted of three pegs attached to a base and a tower of eight disks of different diameters stacked in order of size on one peg, with the largest on the bottom as shown in Figure 1.1.

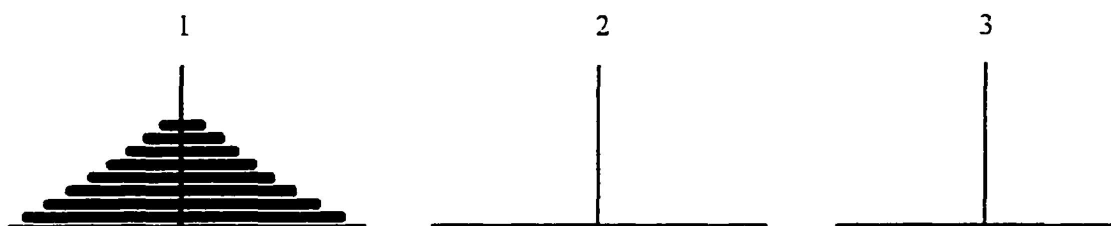


Figure 1.1 Towers of Hanoi -- The Initial Configuration for 8 Disks

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The object of the puzzle is to transfer all of the disks in the same order onto another peg (see Figure 1.2) abiding the following restrictions:

1. Only the topmost disk can be moved from each peg.
2. A larger disk cannot be placed on a smaller disk.

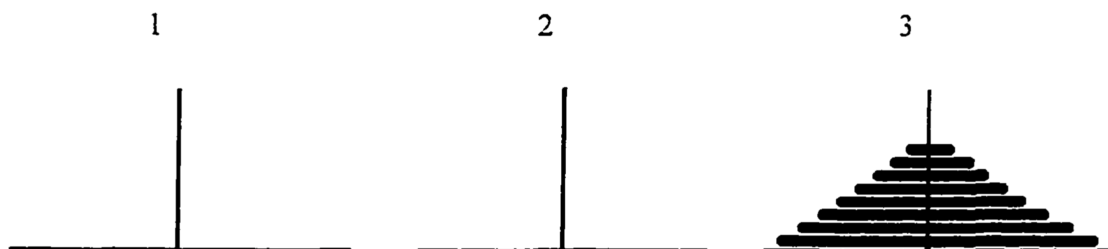


Figure 1.2 -- The Final Destination for 8 Disks

In 1884, H. de Parville [41] revealed "Claus" to be the anagram/pseudonym for "Lucas" -- and the name N. Clause de Siam to be a scramble of "Lucas d'Amiens" (Amiens was Lucas' birthplace) [43, 26]. Accompanying this unmasking was de Parville's eschatological revelation of the mythical Tower of Brahma:

> In the great temple at Benares, beneath the dome which marks the center of the world, rests a brass plate in which are fixed three diamond needles, each a cubic high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four disks of pure gold, the largest disk resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the Tower of Brahma. Day and night unceasingly the priests transfer the disks from one diamond needle to another according to the fixed and immutable laws of Brahma, which require that the priest on duty must not move more than one disk at a time and that he must place this disk on a needle so that there is no smaller disk below it. When the sixty-four disks have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish [52].
>
> M. de Parville
>
> La Nature, 1884

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It has been put forward by A.M. Hinz [26] that Lucas himself likely invented the Towers of Hanoi puzzle, named for a city current in the French newspapers of the day as the site of colonial military excursion. De Parville's later implication that the Towers of Hanoi was based on legend contributed to its fame, wonder and popularity [43]. Lucas, was already marketing the puzzle in a box bearing the name of a legendary, ancient Chinese emperor, Fu Xi, whom he also credited with the invention of the binary number system. Lucas also included an insert describing the puzzle as having been found among the papers of the "illustrious Mandarin Fer-Fer-Tam-Tam," a risible reference to Fermat, as an anagram, and also, possibly, in the doubled syllables, to binary numbers. Fermat had not referred to the puzzle, though Lucas might have been jocularly poking Fermat's mistakenly asserted prime number $2^{64} + 1$ by reversing the second syllable of Fermat's name (just as reversing the purported prime's plus sign yields the optimal number of moves to solve the puzzle for 64 disks: $2^{64} - 1$). Lucas, it will be remembered, was a past master of prime number calculation and, as a compiler of Fermat's papers, was surely familiar with Fermat's letter to Frenicle, putting forward this prime number candidate [26].

That Lucas should have propounded the puzzle is not surprising, as it illustrates a problem that could be solved by the prize winning mathematics of Lucas' older contemporary, Sir Wm. Rowan Hamilton. The literature is replete with acknowledgments of Crowe's discovery [9] that the solution of the Towers of Hanoi puzzle is equivalent to finding a Hamiltonian path on an n-cube [26, 43, 14]. Lucas' reference to China might have been a tip-off to the related solution of the popular Chinese Ring Puzzle [19].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Aside from its popularity among puzzle enthusiasts, the Towers of Hanoi puzzle frequently appears in computer science textbooks to demonstrate comparative programming [33, 44] and in writings on algorithms [42], discrete mathematics [37], artificial intelligence [8] and combinatorics [6]. Cull and Ecklund [10] use it to discuss complexity of algorithms. It is even employed in the psychological testing of human decision making [49].

1.4 Solutions and Extensions

In 1884, the first solution to the Towers of Hanoi puzzle, proposed by Allardice and Fraser [2], appeared in the mathematical literature. To calculate the number of moves necessary to transfer the stack of n disks from one peg to another, we will follow the steps in Figures 1.3(a) through (d):

- Let H_n denote the total number of moves for this transfer.

Figure 1.3(a) TOH -- Initial Configuration for n Disks

- Transfer the top n-1 disks from the initial peg to an intermediate peg, one at a time; this transfer requires H_{n-1} disk moves.

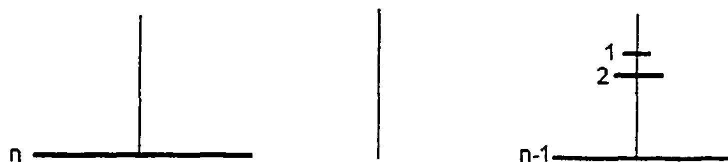


Figure 1.3(b) Transfer of the Top n-1 Disks

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Transfer the largest disk, n, to the target peg; this requires only one move.

Figure 1.3(c) -- Transfer of the nth Disk to the Target Peg

- Transfer the remaining n-1 disks from the intermediate peg onto the target peg in a similar fashion; this transfer will take H_{n-1} disk moves.

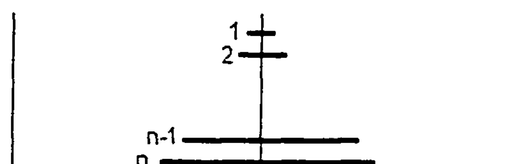


Figure 1.3(d) -- Transfer of n- 1 Disks to the Target Peg

Remark:

As Allouche [3] points out, "there are *two* different solutions" to the Towers of Hanoi puzzle, depending upon whether n, the number of disks, is odd or even. If n is odd, the transfer is made from initial peg 1 to the target peg 2, but if n is even, the transfer is made from initial peg 1 to target peg 3.

It is clear that transferring n disks (for $n > 0$) will require $2H_{n-1}+1$ disk moves.

Hence,

$$H_n = 2H_{n-1}+1 , \qquad \text{for } n > 0 \quad \text{and with the initial condition } H_0 = 0.$$

This relation, characterized by an equation, a boundary value and an initial condition, is called a *recurrence* relation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In the recurrence relation we consider several values of n, in which $n > 0$:

$$H_1 = 2 \times 0 + 1 = 1$$

$$H_2 = 2 \times 1 + 1 = 3$$

$$H_3 = 2 \times 3 + 1 = 7$$

$$\ldots$$

it is easily seen that for $n \geq 0$,

$$H_n = 2^n - 1$$

is the optimal solution. There are 3^n possible configurations for H_n [43].

As David Poole [43] has pointed out, this was not known to be the optimal solution until it was established by Wood [58] in 1981, who also demonstrated the uniqueness of the optimal sequence of moves.

"It is hard to see how such a simple problem with a complete and well known solution could possibly hold anything more for us. But as we shall see, Professor "Claus," perhaps unintentionally, built so much into his little puzzle that we are just at the beginning of the story." [43]

The "little puzzle" is inspiring intensive research. Recent explorations of the Towers of Hanoi graph have delved into its resemblance to the Sierpiński gasket (a fractal graph) and its correspondence to Pascal's triangle [43, 52]. Poole has used this *Lucas Correspondence* [43] to solve various problems arising from the puzzle. Allouche [3] has demonstrated that the puzzle generates an infinite squarefree sequence, a sequence in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

which no move is immediately repeated. Fournier has devised a 2-automaton for finding the *n*th move of this puzzle [4].

The following are some of the many variations that have been developed since the puzzle's appearance over one hundred years ago:

Superhanoi [30], in which the disks in the initial configuration are distributed among the pegs in an arbitrary manner; larger disks can be atop smaller ones.

Rainbow Hanoi [38], in which the disks have various colors and their movements are restricted.

Cyclic Hanoi [5], in which the pegs are arranged in a circular order and the disk movements are either made clockwise or counterclockwise [16, 17, 18].

Straightline Hanoi [45], in which the disks may only be moved to or from peg 1.

Multi-peg Hanoi [37], in which more than 3 pegs are used.

Multi-disk Hanoi [58], in which several copies of the same disk is allowed.

The observation that has encouraged us to use the Towers of Hanoi puzzle to generate an encryption algorithm, is that each move brings the puzzle to a new and unique configuration. By coding these moves, the resultant strings can be used in the design of a cryptosystem.

Let us now focus on the classical Towers of Hanoi puzzle to derive a cipher.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 2

2.1 Definitions and Terminology

To insure secure communication between a *sender* and a *receiver*, a message must be disguised. Before it is disguised, the message or text is called *plaintext* or *cleartext*. The encrypted message is called *ciphertext*. The sender and the receiver share a *key*. The sender, using the key, converts the plaintext to ciphertext. The procedure of disguising the content of a message is called *encryption* or *encipherment*. The receiver, using the key, will convert the ciphertext to its original plaintext form. This conversion process is called *decryption* or *decipherment*. The practice of these writing techniques constitutes *cryptography*. Senders and receivers who employ these techniques are *cryptographers*. The practice, by unauthorized users, of attacking ciphertexts, is called *cryptanalysis*. The attackers are known as *cryptanalysts*. *Cryptology* encompasses both cryptography and cryptanalysis. Practitioners of cryptology are called *cryptologists* [46, 55].

A *cryptosystem* is designed to encrypt plaintexts and decrypt ciphertexts using keys and an algorithm. See Figure 2.1. A *cryptosystem* consists of :

1. M, a finite set of possible *plaintexts* from a finite alphabet Σ^*,
2. C, a finite set of possible *ciphertexts* from a finite alphabet Σ^*,
3. K, a finite set of possible *keys*,
4. $e_k \in E$, an *encryption algorithm*, where, $k \in K$ and $e_k : M \rightarrow C$ a function,
5. $d_k \in D$, a *decryption algorithm*, where $k \in K$, and $d_k : C \rightarrow M$ a function, such that for every plaintext $m \in M$ and $k \in K$, applying the functions noted above will reveal the plaintext $d_k(e_k(m)) = m$ [55].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Note: Σ denotes an alphabet consisting of a finite set of letters or symbols. A string $m = m_1 m_2 \ldots m_n$ is an ordered sequence of symbols from Σ. The collection of all finite strings from Σ is denoted by Σ^* [56].

To communicate, the sender and the receiver, have to agree in advance on a key, k, a member of the set of possible keys K (represented as: $k \in K$). The sender will use the key to encrypt a plaintext string $m \in M$, (where $m = m_1, m_2, \ldots, m_n$ and $n \geq 1$) by applying the encryption algorithm e_k to each m_i for $1 \leq i \leq n$:

$$c_i = e_k(m_i).$$

The resulting string $c = c_1, c_2, \ldots, c_n$ is the ciphertext to be sent.

The receiver, with the key in hand, applies the decryption algorithm d_k to each $c_i \in C$ for $1 \leq i \leq n$:

$$m_i = d_k(e_k(m_i))$$

to decrypt the ciphertext string $c_1, c_2, \ldots, c_n$ and reveal the original plaintext:

$$m = m_1, m_2, \ldots, m_n.$$

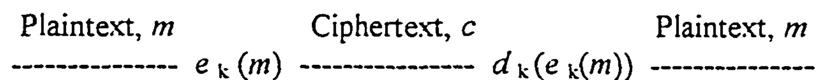


Figure 2.1 A Cryptosystem

Ciphers are commonly based upon modular arithmetic.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Definition:

Suppose a and b are integers, and m, called the *modulus*, is a positive integer. Then $a \equiv b \pmod{m}$ if $a = b + km$ where k is an integer, a, is a non-negative integer and $0<b<m$ and can be thought of as the remainder of a when divided by m. The sign, ≡, denotes *congruence*, and $a \equiv b \pmod{m}$ can be read " a is congruent to b mod m" or " b is residue of a modulo m" [46, 55].

Example:

Let $a = 17$, $b = 11$, and $m = 26$.

Then (17 + 11) mod 26 = 28 mod 26 = 2 mod 26

Or, using the congruence relation:

$17 + 11 \equiv 2 \pmod{26}$.

Table 1, which shows the correspondence between alphabetic characters and modulo 26, will be used throughout in encryption examples:

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25

Table 1. Number-Alphabet Correspondence

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2 The Vigenere Cipher

This cipher is named for Blaise de Vigenère, a sixteenth century diplomat of France and student of cryptography. In this system the letters of the alphabet A, . . ., Z are given a numerical value in the range 0, . . ., 25. The key k consists of an alphabetic string of length d, called the *keyword*. The keyword is written repeatedly below the plaintext and modulo 26 addition is performed on the keyword letters and plaintext. This cryptosystem is called *polyalphabetic*, since a plaintext character is mapped by one of the several possible alphabetic characters defined by d [55].

It is harder for cryptanalysis to break a polyalphabetic cryptosystem than a *monoalphabetic* one in which there is a fixed correspondence between the letters of the plaintext and the letters of the key [55, 56].

Example:

Let $d = 5$, and the *keyword* = CRYPT. The corresponding numerical values for this keyword are k = (2 17 24 15 19).

Suppose the plaintext string is m ≡ WORLDPEACE

and k ≡ CRYPTCRYPT

modulo 26 addition yields c ≡ YFPAWRVYRX.

Vigenere proposed, as a refinement to strengthen his cipher, an autokey system, in which a keyword is concatenated with the plaintext itself to provide a running key. This autokey system is also vulnerable. Strong encryption would result from using a random

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sequence of key letters (which constitutes the key *k*) as long as the length of the plaintext. This is what Gilbert Vernam proposed in 1918. His keyword was carried on a loop, which eventually repeated, and led, almost immediately, to Joseph Mauborgne's modification. He proposed, as an improvement, altering each letter of plaintext by the corresponding letters of a non-repeating, random key of equal length. This is known as a one-time pad and is generally proving to be unbreakable if the sequence is random [51, 55, 46].

2.3 One-Time Pad

A one-time pad originally consisted of a large nonrepeating set of random letters glued to a pad. It has been called a one-time tape (when the key is stored on magnetic tape). It was first used in the transmission of encrypted teletype messages [46]. It is the classic example of a perfectly secure cryptosystem; virtually the only such example unless the plaintext is restricted to be somehow random. Both the sender and the receiver hold identical copies of the pad. Each key letter on the pad is used to encrypt one corresponding plaintext character. The encryption operation is achieved by a modulo 26 addition of the plaintext and the one-time pad key characters. Each pad is used only once. Upon the completion of the encryption the sender destroys the pad. The receiver uses the pad to decrypt the message and then destroys the used key letters [56, 55].

Example:

If the plaintext message is: SECURE

and the key letters on the pad are: LESPGT

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

performing mod 26 addition upon each character yields:

S + T mod 26 = L

E + E mod 26 = I

C + S mod 26 = U

U + P mod 26 = J

R + G mod 26 = X

E + Y mod 26 = C

The ciphertext obtained is: LIUJXC.

The disposable random key makes cryptanalysis of the ciphertext impossible for the cryptanalyst, if the key is "truly random". In this scheme the length of the pad runs at least the length of the plaintext. Though this is the strength of the one-time pad cryptosystem, it is cumbersome and therefore considered a drawback. Generating a one-time pad from the Towers of Hanoi puzzle should greatly facilitate key exchange by enabling a sender to transmit a puzzle's characteristics and configuration as an abbreviated key which the receiver could then derive.

2.4 Shift Registers and Finite State Machines

2.4.1 Introduction

Shift registers and their sequences have been frequently used in coding theory and the design of cryptosystems. They are simple in design, easy to build and offer a high level of security. Most military encryption systems still in use are based on linear feedback shift registers (LFSRs) [46].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Since shift registers produce random sequences of 0's and 1's, their sequences can be used as a key for encryption, where the key is added to a binary plaintext message using mod 2 operation; the Vigenère cipher is an example. Shift register sequences are also used to authenticate transmitted messages [21].

Shift registers can be described as examples of finite state machines, which we now briefly discuss.

2.4.2 Finite State Machines

A finite state machine M, consisting of a finite set Q of *internal states*, which accepts a finite set X of *input symbols*, produces a finite set Y of *output symbols* and then changes its state [21, 32]. It is further comprised of an *output function* λ which computes the present output as a fixed function of present input and present state:

$$\lambda : Q \times X \rightarrow Y,$$

and a *next-state function* μ which computes the next state as a fixed function of present input and present state:

$$\mu : Q \times X \rightarrow Q .$$

Definition:

A finite state machine $M = \langle X, Y, Q, \mu, \lambda \rangle$ where:

(1) X is a finite set of input symbols.

(2) Y is a finite set of output symbols.

(3) Q is a finite set of states.

(4) $\mu : Q \times X \rightarrow Q$ is a next-state function.

(5) $\lambda : Q \times X \rightarrow Y$ is an output function [21, 32].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example:

The finite state machine M is defined by two input symbols, three states and three output symbols as follows:

(1) $X = \{0, 1\}$

(2) $Y = \{a, b, c\}$

(3) $Q = \{q_0, q_1, q_2\}$

(4) $\mu : Q \times X \to Q$ next-state function is defined by

$\mu(q_0, 0) = q_1$ $\mu(q_0, 1) = q_2$

$\mu(q_1, 0) = q_1$ $\mu(q_1, 1) = q_0$

$\mu(q_2, 0) = q_0$ $\mu(q_2, 1) = q_1$

(5) $\lambda : Q \times X \to Y$, the output function is defined by

$\lambda(q_0, 0) = a$ $\lambda(q_0, 1) = b$

$\lambda(q_1, 0) = a$ $\lambda(q_1, 1) = c$

$\lambda(q_2, 0) = c$ $\lambda(q_2, 1) = b$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

There are two ways of representing finite state machines [21]:

- by a *function table*, which for every possible combination of state and input has corresponding output and next state function values. An example of this representation is illustrated in Figure 2.2.

present state	μ, λ X=0	X=1
q_0	q_1, a	q_2, b
q_1	q_1, a	q_0, c
q_2	q_0, c	q_1, b

Figure 2.2 Function Table of a Finite State Machine.

- by a *state diagram*, which uses a labeled directed graph (see Figure 2.3).

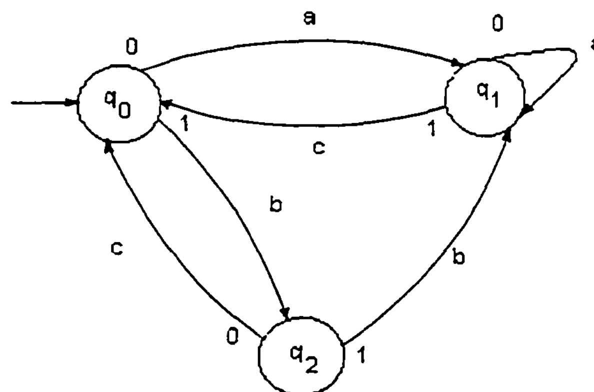


Figure 2.3 A State Diagram for a Finite State Machine.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The *behavior* of a finite state machine depends solely upon the various input sequences, e.g.:

- An infinite number of 0's;
- An infinite number of 1's;
- 1 followed by an infinite sequence of 0's;
- Finite number of 1's followed by infinite number of 0's.

When an input consists of a finite non-zero portion followed by zeros, the output of the machine during the zero portion is called the *autonomous* behavior of the machine [21].

As Golomb points out, according to an important theorem, the output sequence of all finite state machines for all input combinations of the strings listed above "will ultimately become (and remain) periodic."

2.4.3 Shift Registers

A shift register consists of a collection of n registers (or binary storage elements) arranged in a row labeled $x_1, x_2, \ldots, x_n$; with the registers storing either 1 or 0 (see Figure 2.4). At periodic intervals controlled by a clock, the content of each register is shifted to the next register.

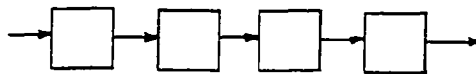


Figure 2.4 A Shift Register

2.4.4 Linear Feedback Shift Registers

In an n-bit *feedback shift register*, at each clock pulse, the content of register x_i is transferred into x_{i-1}. The new value for the leftmost register x_1 is computed using a feedback function $f(x_1, x_2, \ldots, x_n)$ of the other bits in the shift register. The output of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the shift register is produced one bit at a time at the point of the least significant bit (see Figure 2.5).

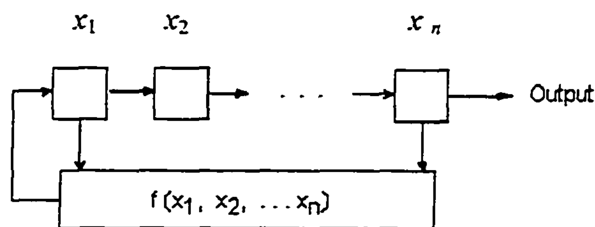


Figure 2.5 A Feedback Shift Register

The periodicity of a shift register occurs one step before the output sequence repeats. The length of the output sequence is thus the shift register's period. Shift registers (linear or nonlinear) with n bits have a maximum period of 2^n-1 [21, 56].

A *linear feedback shift register* (LFSR), XORs the content of certain registers and feeds it back into the leftmost register (x_1).

Example:

Suppose we have a 3-bit shift register as shown in Figure 2.6:

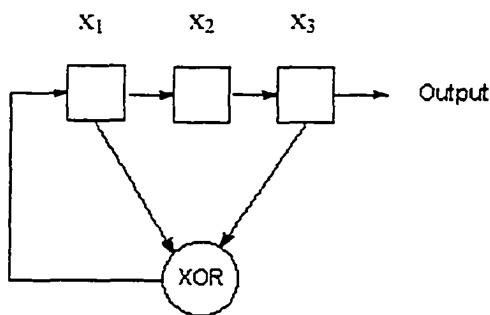


Figure 2.6 A 3-bit Linear Feedback Shift Register.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

If the initial value of the shift register is 111 and the content of registers x_1 and x_3 are chosen to be XORed and fed back to x_1, then the sequence of internal states would be as follows:

111

011

101

010

001

100

110

111

As we notice, the first and the last states have the same sequence and therefore, a period occurs at step $7 = 2^3 - 1$. If we take the least significant bit of every string, before the repetition begins, we will have the following output sequence: 1110100.

In shift registers, the n registers are referred to as the *stages* of the shift register, and their content is called the *state* of the shift register. At every pulse, a shift occurs, and a transition from one state to the next takes place. For an n-stage shift register there are 2^n possible states. Eventually, after going through some sequence of states, an autonomous behavior will occur, yielding a periodic succession.

Thus, an n-stage general shift register can be described as a finite state machine with 2^n possible states [21].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.5 Stream Ciphers

Stream ciphers, in which a keystream generator (also known as running key generator) produces a *keystream* (or running key), are common in modern cryptography. A keystream, $k = k_1, k_2, \ldots, k_i$, is used at both the sending and receiving end.

The keystream elements are XORed with the plaintext bits $m = m_1, m_2, \ldots, m_i$ according to the encryption rule to produce ciphertext:

$$c = c_1, c_2, \ldots, c_i = e_k(m_1), e_k$$

$(m_2), \ldots, e_k(m_i)$ [45, 54].

Decrypting a stream of ciphertext bits can be accomplished by performing an XOR operation on both the ciphertext bits and the keystream bits, one at a time, to recover the plaintext:

$$m = m_1,\ m_2,\ \ldots,\ m_i = d_k(e_k(m_1)),\ d_k(e_k(m_2)),\ \ldots,\ d_k(e_k(m_i)).$$

Stream ciphers operate on binary alphabets, zeros and ones. For this reason the encryption and decryption is accomplished using XOR, which is bitwise add mod 2 [46, 55] (see Figure 2.7).

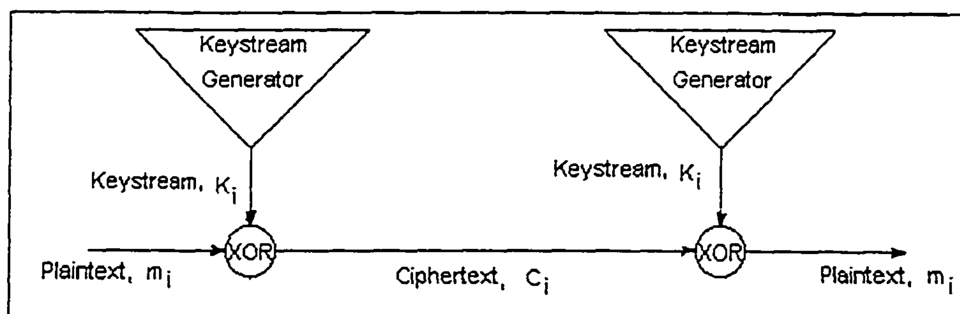


Figure 2.7 A Stream Cipher [46]

The output of the keystream generator controls the security of the system [46]. If the output string has repeating bit patterns, the security level is low. However, if a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

keystream can be generated which is comprised of a random sequence of plaintext length, it offers the perfect secrecy of a one-time pad cryptosystem.

Stream ciphers can use LFSRs in their design. Although LFSR-based stream ciphers are simple to design and easy to build, if well designed, they can be secure. Since LFSRs produce apparently random sequences, one bit at a time, stream ciphers made up of LSFRs remain popular among cryptographers [46, 21].

In designing keystream generators, LFSRs of different lengths and different feedback functions are used. The key becomes the initial state of these LFSRs. The output bits, generated one bit at a time at each clock pulse, are the keystream bits. The more the keystream resembles randomness, the closer it is to a one-time pad and thus to achieving security against the cryptanalyst with overwhelming computing time and power [46].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 3

In this chapter we will present an encryption algorithm focusing on the following concepts:

a) the Towers of Hanoi puzzle (TOH) to produce sequences,

b) quadratic residues,

c) Huffman codes,

d) one-time pad encryption, and

e) stream ciphers for encrypting plaintexts.

We padlock a message with encryption, requiring as a key a one-time **pad** based upon **Lucas**' puzzle, and therefore call this the **Padluc** encryption algorithm. The next sections will describe some of the terms and concepts comprising the Padluc algorithm.

We first define the concept of quadratic residues which will be used by the Padluc algorithm as a tool for sampling sequences generated by the TOH algorithm by allowing $1 \leq x \leq 2^n - 1$ and $1 \leq p \leq 2^n - 1$. The use of conventional pseudo-random number generators contributes to additional risk to any implementation and we avoid it.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.1 Quadratic Residues

If p is a prime, and $0 < a < p$, then a is a quadratic residue mod p if

$$x^2 \equiv a \pmod{p}, \text{ for some } x.$$

This is true for half of all values of a [46].

Example 3.1: Suppose $p = 7$,

$$1^2 = 1 \equiv 1 \pmod{7}$$

$$2^2 = 4 \equiv 4 \pmod{7}$$

$$3^2 = 9 \equiv 2 \pmod{7}$$

$$4^2 = 16 \equiv 2 \pmod{7}$$

$$5^2 = 25 \equiv 4 \pmod{7}$$

$$6^2 = 36 \equiv 1 \pmod{7}.$$

The values 1, 2, and 4 are the quadratic residues, each of which occurs twice.

The values of a which do not appear on the list above are 3, 5, and 6. These values are called **quadratic nonresidues** modulo 7, since there are no values of x that would satisfy any of the following equations [46]:

$$x^2 \equiv 3 \pmod{7}$$

$$x^2 \equiv 5 \pmod{7}$$

$$x^2 \equiv 6 \pmod{7}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Once a sequence is sampled by the above method, the Padluc algorithm will then employ the Huffman codes described in the next section to compress the sequence.

3.2 Huffman Codes:

Huffman codes are a well known and widely used method of compressing data. The construction of the codes employs a greedy algorithm to build a bottom-up tree whose leaves (boxes in Figure 3.1) contain characters and their frequencies in an increasing order from left to right. At each step, the greedy algorithm merges the two trees with the lowest frequencies. The sum is placed in internal nodes (circles in Figure 3.1). Once the final tree has been reached, the edges are labeled, from root to leaf, with 0 (if the edge is a left child of a node) or otherwise, 1. The sequence of edge labels on the path from the root to the leaf on which a character resides, corresponds to that character's codeword.

Example:

Suppose an alphabet of 6 letters with the following frequencies is given:

a:39 b:7 c:3 d:25 e:9,

Applying the Huffman algorithm to these frequencies and following the required steps, we will reach a final tree as shown in Figure 3.1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

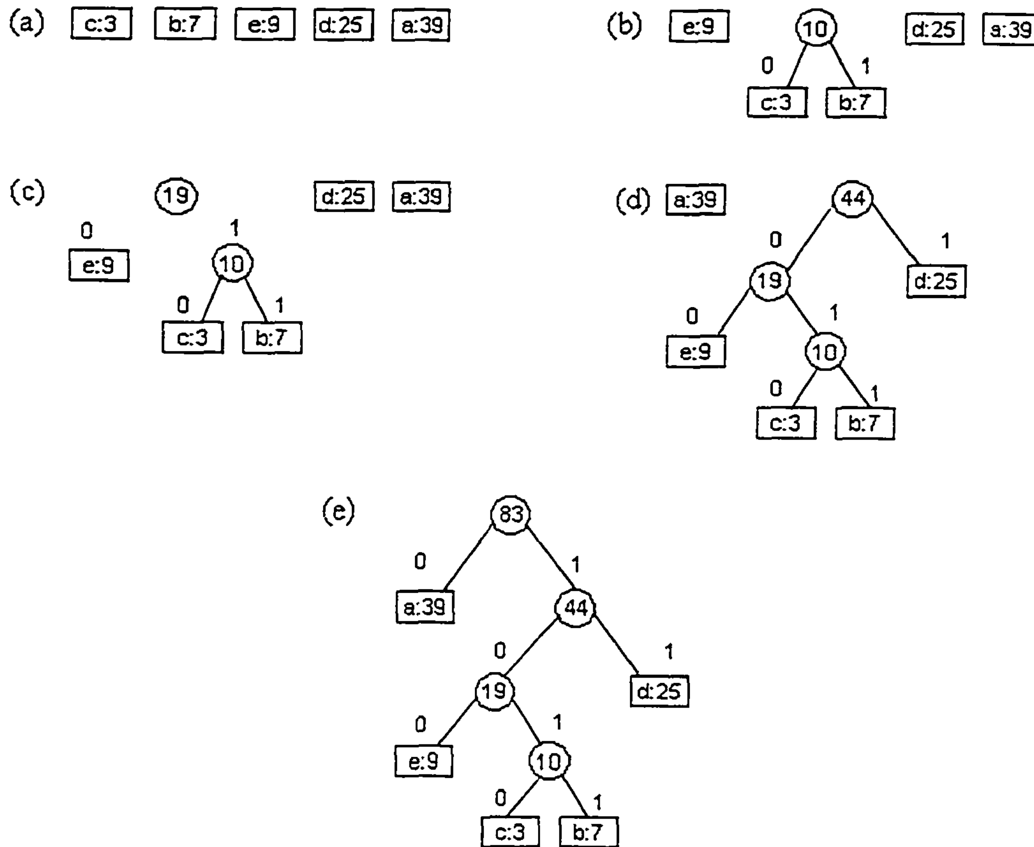


Figure 3.1 (a) The initial nodes, (b) through (d) Intermediate steps, (e) The final tree.

The following codewords are obtained from Figure 3.1(e):

a = 0 b = 1011 c = 1010 d = 11 e = 100

A sequence of characters can be encoded by concatinating the codewords representing each character.

Example:

Given the sequence **d a a e d c**, and the above codewords, the following string yields:

1100100111010

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

An encoded string can be decoded by translating the codewords into their original characters.

Example:

The string **10010100100** yields the original characters:

e c a e

In order to test the Padluc algorithm for randomness, we will use a statistical test called Chi-square.

3.3 Chi-Square Test:

This well known statistical test is used to measure randomness of data.

Suppose there are n independent observations and k categories, every observation falling into one of k categories. By counting the number of observations falling into each of the categories we can calculate the Chi-square value V :

$$V = \sum_{1 \le s \le k} \frac{(Y_s - np_s)^2}{np_s}$$

where p_s is the probability for each observation to fall into category s, and Y_s is the actual number of observations that fall into category s. The value of V is then compared with the values in Table 1 [31], which gives values of "the Chi-square distribution with $v = k - 1$ degrees of freedom" for different values of v.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

	p=1%	p=5%	p=25%	p=50%	p=75%	p=95%	p=99%
ν=1	0.00016	0.00393	0.1015	0.4549	1.323	3.841	6.635
ν=2	0.02010	0.1026	0.5753	1.386	2.773	5.991	9.210
ν=3	0.1148	0.3518	1.213	2.366	4.108	7.815	11.34
ν=4	0.2971	0.7107	1.923	3.357	5.385	9.488	13.28
ν=5	0.5543	1.1455	2.675	4.351	6.626	11.07	15.09
ν=6	0.8720	1.635	3.455	5.348	7.841	12.59	16.81
ν=7	1.239	2.167	4.255	6.346	9.037	14.07	18.48
ν=8	1.646	2.733	5.071	7.344	10.22	15.51	20.09
ν=9	2.088	3.325	5.899	8.343	11.39	16.92	21.67
ν=10	2.558	3.940	6.737	9.342	12.55	18.31	23.21
ν=11	3.053	4.575	7.584	10.34	13.70	19.68	24.73
ν=12	3.571	5.226	8.438	11.34	14.84	21.03	26.22
ν=15	5.229	7.261	11.04	14.34	18.25	25.00	30.58
ν=20	8.260	10.85	15.45	19.34	23.83	31.41	37.57
ν=30	14.95	18.49	24.48	29.34	34.80	43.77	50.89
ν=50	29.71	34.76	42.94	49.33	56.33	67.50	76.15
ν>30	$\nu + \sqrt{2\nu}x_p + \frac{2}{3}x_p^2 - \frac{2}{3} + o(1/\sqrt{\nu})$						
x_p =	-2.33	-1.64	-.675	0.00	0.675	1.64	2.33

Table 1. Chi-square Distributions [31]

If the value of V lies between the 0-1 percent entry or the 99-100 percent entry, its randomness is “rejected.”

If the value of V lies between the 1-5 percent entry or the 95-99 percent entry, its randomness is “suspect.”

If the value of V lies between the 5-10 percent entry or the 90-95 percent entry, its randomness is “almost suspect.” [31].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To test the Padluc algorithm for periodicity, we perform a Kasiski test.

3.4 Kasiski Examination:

This test is used to perform the following operations:

- locate repeated substrings in a ciphertext,

- find the distance between the repeated substrings,

-break up the value of the distance into its factors,

The factor most frequently found indicates the number of letters in the key. The cryptanalyst uses this information to decide upon a period likely to give the greatest chance of exposing a repetition [29].

As one of its components, the Padluc algorithm will use a procedure called getSequence that makes possible the task of going from a number to a configuration.

3.5 PROCEDURE getSequence:

We propose this procedure as a tool to allow the Padluc algorithm direct access to Towers of Hanoi configurations without having to solve the puzzle. This algorithm has two inputs: the number of disks, n, and the Towers of Hanoi configuration number, obtained by calculating a quadratic residue value. Performing a recursive operation, this algorithm will obtain the sequence associated with the specified configuration number.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The following is the pseudocode for procedure getSequence:

```
PROCEDURE getSequence(configuration_num, num_disks, from_tower, to_tower,
spare_tower)
  IF (n = 0)
     do nothing
  IF (c < 2^(n-1))    we have not moved the largest disk yet
     OUTPUT from_tower
     getSequence(configuration_num, n-1, from_tower, spare_tower, to_tower)
  ELSE      we have already moved the largest disk
     OUTPUT to_tower
     getSequence(configuration_num - 2^(n-1), n-1, spare_tower, to_tower, from_tower)
```

Finally, we present the layout of the Padluc algorithm.

Note:

Several pieces of information are known only by the sender and the receiver:

1. The tower representations,
2. start_x, the starting point for calculating quadratic residues, and
3. primes p & q.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.6 Padluc Algorithm Layout:

- Get plaintext

- Let n be the number of disks in Towers of Hanoi

- Let p and q be two primes, where $1 \le p, q \le 2^n - 1$,

- Let x = start_x, where start_x is the initial value chosen to calculate quadratic residues

While length of Padluc_key < plain_len, do the following:

{

  - $x = x + 1$
  - Calculate quadratic residues $x^2 \equiv a \pmod{p}$ and $x^2 \equiv b \pmod{q}$
  - Call Procedure getSequence to use the above quadratic residue values as TOH configuration numbers to find sequences associated with them.
  - Apply the Huffman encoding algorithm to the sequences
  - XOR the encoded results
  - Concatenate the result of the previous step with the content of the key

}

- cipher_bin = Padluc_key $\oplus$ plain_bin

-Return

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Note:

The Padluc algorithm will convert the key into a stream cipher in which the key becomes the initial state of the LFSR-based keystream generator. The stream cipher will take the plaintext and the output of the keystream generator, one bit at a time, and perform an XOR operation. The encrypted result of this operation will be our ciphertext.

3.7 Padluc Example:

Let plaintext = STOP; the binary representation would be:

100111010001111110000

n (number of disks) = 6

T1 = A, T2 = B, and T3 = C (tower representations)

p = 61, q = 59

x = start_x = 9

(1) Calculate quadratic residues:

$a = 9^2 \bmod 61 = 20$

$b = 9^2 \bmod 59 = 22$

(2) Call getSequence to find sequences associated with configuration numbers 20 and 22:

- Sequence for configuration number 20 = BBACBA

- Sequence for configuration number 22 = CAACBA

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(3) Huffman encode the above sequences:

Using the Huffman encoding technique described in section 3.2, we will obtain the following codewords for sequence BBACBA:

A = 01 B = 1 C = 00

The encoded sequence would be: 110100101

The codewords for sequence CAACBA are:

A = 1 B = 00 C = 01

The encoded sequence would be: 011101001

(4) XOR the encoded sequences:

The result is as follows: 101001100

(5) Concatenate this result with the content of key_buf:

key_buf = 101001100

(6) Is the key_buf_len $\geq$ plain_len? No

(7) $x = x + 1 = 10$; repeat steps 1 through 5:

key_buf = 101001100001110100

Since key-len_buf < plain_len, we will repeat this process (steps 1 through 5) for $x = 11$ and get:

key_buf = 101001100001110100010000

Is the key_buf_len $\geq$ plain_len? Yes

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

XOR the content of the key_buf and the plaintext to get ciphertext:

Ciphertext = 0011101100000000100001

For a C implementation of the Padluc algorithm, including Chi-square and Kasiski tests, the plaintext used for testing, the test results and a discussion of the test results, see the Appendix. The C implementation of the Padluc algorithm is also presented in the Appendix with three additional test cases and a discussion of results. There are, of course, many Padluc variations; we present just a few.

The problem of breaking the Padluc encryption algorithm is to be dealt with in future research as we appreciate the enormity of a problem which may extend over decades [7].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 4

4.1 Conclusion:

It has been fascinating to discover, in the work of a man pursuing "recreational mathematics" more than a hundred years ago, so much to be utilized in the exploration of cryptology's future.

The Towers of Hanoi puzzle, to many, (including my 6 year old son who, at once, solved the puzzle for three disks) may not seem to hold complexity and challenge. Its deceptive simplicity points elegantly though, to the more complex concepts of the fractal, squarefree sequences, Pascal's triangle and the Hamiltonian circuit. It has drawn continuous interest for a century and inspired the creation of numerous variations. A very recent variation concerning complexity and cognition appears in the literature [57]. This dissertation, which demonstrates the feasibility of utilizing the Towers of Hanoi to generate one-time pads, may signal the beginning of research into the puzzle's practical application.

We have here proposed to "one-time padlock" a plaintext message with an encryption algorithm and use the Towers of Hanoi puzzle to make a key.

With his puzzle in hand, "Professor Claus," an adept at binary numbers, who gazed so deeply into the primes and series which were to become the tools of codemakers, might himself have taken this small step of creating a cipher to replace the venerable Vigenère cipher, already vulnerable in his day [29].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2 Recommendations for Further Study

The following topics suggest fruitful paths for continued research into the relationship between the Towers of Hanoi problem and problems of security.

4.2.1: Towers of Hanoi, Huffman Code and Authentication

Cryptanalysis of ciphertexts is not the only concern in confidential communications. The importance of transmitting messages without alteration and of confirming the identities of senders, has given rise to the design of many authentication systems for providing *message integrity* and *user authenticity*, among which are, *smart cards* [46].

Having studied the Towers of Hanoi sequences and a paper [20] on the use of Huffman code for encryption, an idea for the authentication scheme came to mind. The researchers documented that to decrypt a Huffman encoded three-symbol source alphabet is "surprisingly difficult."

> In this correspondence we investigate the problem of cryptanalyzing a message that has been compressed using the Huffman algorithm, but not otherwise encrypted. The Huffman algorithm assigns to each source symbol a binary codeword. This arrangement, or *encoding rule*, determines the *codeword set*, which is prefix-free and so corresponds to a full binary tree called the *Huffman tree*. Edges in the Huffman tree connecting an internal node with its left child are labeled 0, and edges connecting an internal node to its right child are labeled 1. The codeword associated with a source symbol is the binary string obtained by reading the bits on the unique path from the root of the tree to the leaf labeled with that source symbol [20].
>
> The source produces a message called the *source stream*, and the encoded version of the source stream is called the *(binary) file*. We show that depending on what the cryptanalyst knows a *priori* about the source and the Huffman encoding, the file can be impossible to decode unambiguously [20].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To see the case of ambiguity, consider the three symbol source alphabet { A, B, C}. In the source stream BABABAACACAA the most common symbol is A, so the Huffman algorithm will assign A the shortest codeword; for example, it may use the encoding rule

A→ 1, B →01, C →00.

The encoded file would then be 01101101110010011. But this file is also an encoding of the source stream ABABABCACAB, using the equally valid encoding rule

A→ 0, B →11, C 10.

So the file is ambiguous [20].

Authentication Scheme:

For the purpose of authentication, the above notion is applied to the Towers of Hanoi sequences.

Example:

Solve the Towers of Hanoi problem for three disks and three towers, using a three-symbol source alphabet {A, B, C} to label the towers A, B, and C. The moves in the Towers of Hanoi are coded as follows:

A → B = AB

B → C = BC

A → C = AC

C → B = CB

B → A = BA

C → A = CA

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

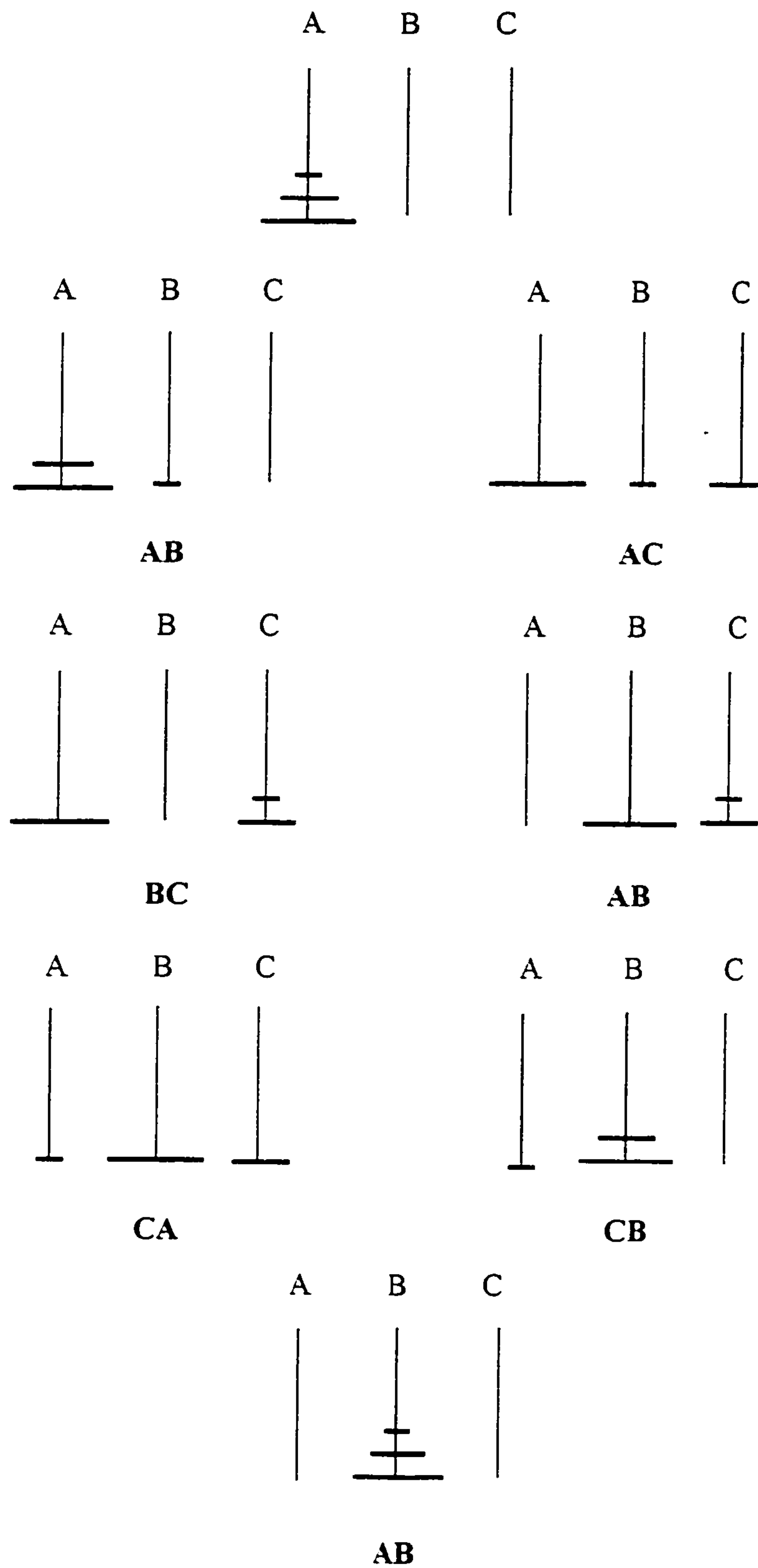


Figure 4.1 Coded 3-Disk Towers of Hanoi

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Concatenate the codes to produce a Hanoi sequence (the source stream [20]):

ABACBCABCACBAB

Counting the number of A's, B's and C's yields 5 A's, 5 B's, and 4 C's.

Place these symbols in a Huffman tree, in an increasing order with respect to their frequencies, from left to right. Since two symbols B and C have equal frequencies, they are placed on two different trees in two different locations, as illustrated in Figures 4.2 (a) and 4.2 (b):

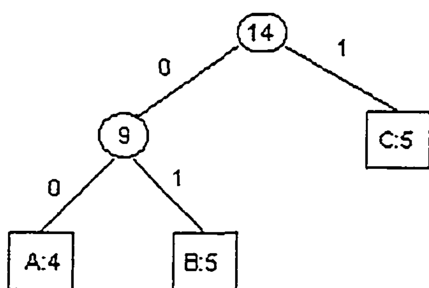


Figure 4.2(a): Huffman Tree

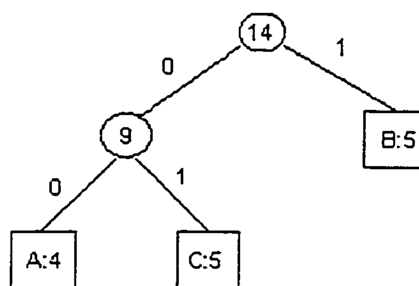


Figure 4.2(b): Huffman Tree

The Huffman compression algorithm is applied to the trees in the following way:

At each step, starting from the bottom, the two frequencies with the lowest value are merged and the result is placed in a circle above them. This process continues until all the frequencies have been covered and we are at the top node of the tree. Upon completion, the tree branches are labeled from top to bottom, 0 on the left and 1 on the right. The codeword for each symbol is the sequence of labels (0s and 1s) from the top node of the tree to the bottom leaf holding the symbol.

The Huffman algorithm assigns the shortest codeword to the most frequently used symbol. Therefore, the encoding rule for the Huffman tree symbols in Figure 4.2(a) is:

$$A \rightarrow 00 \quad B \rightarrow 01 \quad C \rightarrow 1 \quad (I)$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and in Figure 4.2(b) is:

$$A \rightarrow 00 \qquad B \rightarrow 1 \qquad C \rightarrow 01 \qquad (\text{II}).$$

The encoded file using encoding rule (I) would be:

0001001011000110010100001

But this file is also an encoding of the source stream

ACABCBACBABCAC

using encoding rule (II), thus demonstrating the ambiguity.

The solution to the Towers of Hanoi for most values of n (and even some variations) generates equal counts for two of the source stream symbols. Hence, the ambiguity occurs as we demonstrated.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2.2: The Towers of Hanoi Automaton and Encryption Application

To achieve successive movements, Fournier proposed a "machine" [4] which is close to a concrete realization of the finite automaton shown in Figure 4.3.

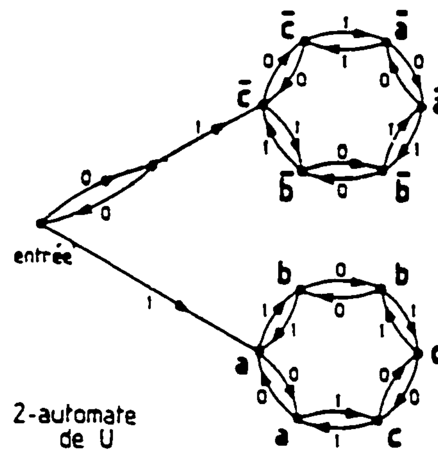


Figure 4.3 2-Automaton for Finding the *n*th TOH Move [4]

The function of the automaton is to give the *n*th movement, which requires investigation of the binary development of n. To calculate the infinite sequence H(n), start with the initial premise and progress to the picture of the automaton by reading, from right to left, the binary development of n and following the arrow 0 or the arrow 1 according to the number read until the automaton furnishes the desired value H(n).

Example:

The equation $(14)_{10} = (1110)_2$ furnishes the value of a.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Encryption Application:

The 2-automaton can be used as a tool in designing a cryptosystem. This can be done by selecting an nth move at random and using its binary representation to encrypt a plaintext character that has also been translated to binary. This process can be repeated to correspond to any length of plaintext. The result is the key. The key and the plaintext can become inputs to a stream cipher to generate a ciphertext.

Example:

If the plaintext is LUC and its binary representation is 011001010100011 and we choose the following moves at random to cover the length of the plaintext:

$(12)_{10} = (1100)_2$, $(26)_{10} = (11010)_2$, $(5)_{10} = (101)_2$, and $(19)_{10} = (10011)_2$.

Concatenation of the binary strings provides:

key = 1100110101011001 1

Key and plaintext, given as inputs to a stream cipher and XORed, will result in ciphertext:

key 11001101010110011
$\oplus$
plaintext 011001010100011

ciphertext 101010000001111

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2.3: The Super Towers of Hanoi (Superhanoi) -- Digital Signature

In this variation the initial distribution of disks on the three towers is completely arbitrary (large disks on small disks are allowed) [30], Figure 4.4.

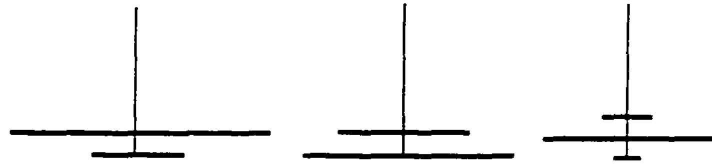


Figure 4.4 The Initial Distribution of Disks in a Superhanoi Puzzle

Designating one of the three towers as the destination tower and following the standard Hanoi rules will bring about the classical Hanoi configuration, Figure 4.5.

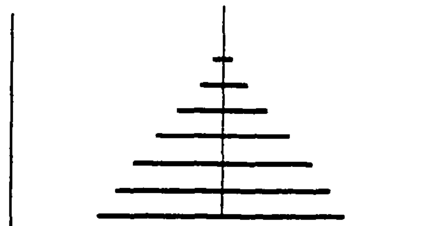


Figure 4.5 The Destination Tower

Klein and Minsker [30] discuss the several possible cases listed below:

Case 1. Largest disk is on the bottom of the destination tower for any initial configuration.

Case 2. Largest disk is not found on the destination tower initially.

Case 3. Largest disk is on destination tower but not on the bottom.

Worst case. All disks are initially on the destination tower in the classical Hanoi configuration, but with disks n and n-1 interchanged.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Digital Signature Scheme

To create a digital signature scheme using the Superhanoi variation, label the towers with single or multiple symbols of an alphabet Σ and solve the problem for an arbitrary initial configuration and destination tower. Code the moves. Solve the problem to generate a sequence. Store the solution in a file. Choose a new initial configuration, destination tower and tower labels. Repeat the process. Once we have a large enough selection of choices, solutions randomly picked from this file can be used as a digital signature and discarded after each use.

Note:

The Superhanoi sequences can also be used to generate a key for the design of an encryption scheme based on the Padluc cryptosystem. The fact that the initial condition is arbitrary contributes to the generation of a key that is difficult to predict.

4.2.4: The Hanoi Graph, Fractals and Password Security

Consider an n-disk Hanoi problem, where disks are labeled 0, 1, . . . , n-1 (0 = smallest, n-1 = largest) and the pegs are labeled A, B, and C. A configuration of disks where larger disks are not on top of smaller disks is called legal and corresponds to an n-bit ternary string $b_{n-1} \ldots b_1 \; b_0$ where the largest disk corresponds to the leftmost bit (b_{n-1}) and the smallest disk to the rightmost bit (b_0). Each $b_i \in \{0, 1, 2\}$ denotes the tower for a disk. Since each disk move results in a unique configuration, the solution will generate 3^n configurations, giving rise to 3^n ternary strings [43].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Definition:

Hanoi graph H_n is a graph whose vertices are labeled by n-bit ternary strings corresponding to the legal configurations in an n-disk Towers of Hanoi problem. In this graph two vertices are adjacent if one can be obtained from the other by a legal move [43]. As demonstrated in Figure 4.6, the labels for each vertex in graph H_3 is the position of the n disks on the three pegs which corresponds to one of 3^n configurations.

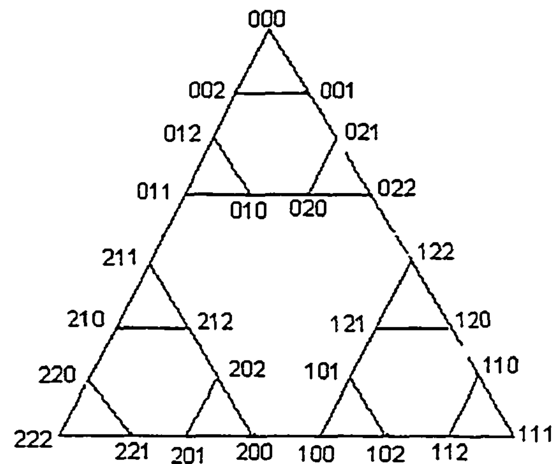


Figure 4.6 The Hanoi Graph H_3

Wacław Sierpiński, a mathematician whose grave bears the Polish words: *Explorer of the Infinite*, invented a curve known today as a *fractal* (called a *Sierpiński gasket* by Mandelbrot [36] in describing a triangular fractal) [52], Figure 4.7.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

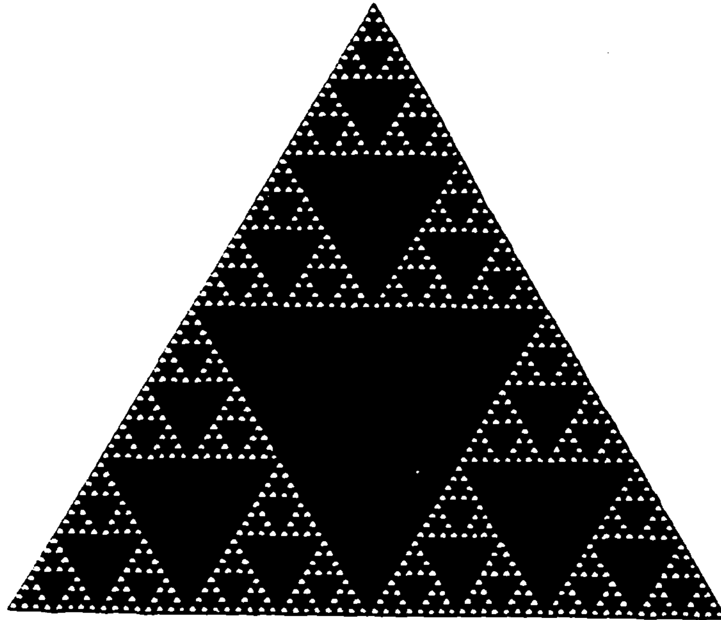


Figure 4.7 The Sierpiński Gasket [52]

As n gets larger, the Hanoi graph H_n becomes a fractal resembling the Sierpinski gasket [52] and Pascal's triangle modulo 2 [43]. The three triangles have fractal dimension $\log_2 3$, Figure 4.8.

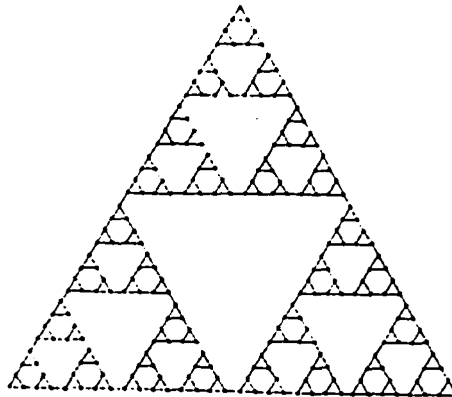


Figure 4.8 The Hanoi Graph H_5 for 5 Disks [52]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Fractal Based Password Security

The Towers of Hanoi fractal could be used for the purpose of password security, by requiring an authorized user to choose a solution path in the graph H_n and prohibiting backtracks. As the fractal graph offers infinite solution paths, finding the correct one would be a difficult task for the unauthorized user especially with the backtrack restriction.

Another possibility for password design using the fractal graph would be to choose certain vertices of the graph at random.

4.2.5: Morphisms and the TOH Squarefree Sequences -- Encryption Application

It has been proved, using the theory of iterated morphisms, that the optimal solution to the Towers of Hanoi puzzle always generates a squarefree sequence -- an *infinite* squarefree sequence for an infinite number of disks [3].

Definition:

Given a finite alphabet Σ, a map h that assigns a string $h(t)$ to each symbol t of the alphabet Σ is called a *morphism*. Σ^* denotes the set of all finite strings of symbols chosen from Σ, using the rule $h(xy) = h(x)h(y)$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example:

Suppose $h(0) = 10$, $h(1) = 01$,

then $h(010110) = h(0)\,h(1)\,h(0)\,h(1)\,h(1)\,h(0) = 10\ 01\ 10\ 01\ 01\ 10$.

If for every symbol, $s \in \Sigma$, the strings of $h(s)$ have a constant length, k, then h is called a *k-uniform* morphism on the alphabet Σ. So, in the above example, we have a 2-uniform morphism φ on the alphabet $\{0,1\}$.

Example :

A finite alphabet $\Sigma = \{a, b, c, \bar{a}, \bar{b}, \bar{c}\}$ is given, a *1-uniform* morphism σ on the alphabet Σ is defined as follows:

$$\sigma(a) = b \qquad \sigma(\bar{a}) = \bar{b}$$

$$\sigma(b) = c \qquad \sigma(\bar{b}) = \bar{c}$$

$$\sigma(c) = a \qquad \sigma(\bar{c}) = \bar{a}$$

and their inverses:

$$\sigma^{-1}(a) = b \qquad \sigma^{-1}(\bar{a}) = \bar{b}$$

$$\sigma^{-1}(b) = c \qquad \sigma^{-1}(\bar{b}) = \bar{c}$$

$$\sigma^{-1}(c) = a \qquad \sigma^{-1}(\bar{c}) = \bar{a}.$$

The relation between this morphism and the Towers of Hanoi puzzle can be described as follows:

"If a string x has the effect of moving some disks from peg 1 to peg 2, using an intermediate storage, it moves the same configuration of disks from peg 2 to peg 3, using peg 1 as intermediate storage" [3].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In solving the Towers of Hanoi puzzle for n disks, disks are moved from peg 1 to peg 2, from peg 2 to peg 3, or from peg 1 to peg 3; and vice versa. Symbols a, b, c, $\bar{a}$, $\bar{b}$, and, $\bar{c}$ are used to code each move respectively.

Example:

The solution for a 4-disk Towers of Hanoi puzzle generates a sequence, which has been coded as follows:

$$h_4 = a\,\bar{c}\,b\,a\,c\,\bar{b}\,a\,\bar{c}\,b\,\bar{a}\,c\,b\,a\,\bar{c}\,b.$$

However, the sequence above contains the following solution for $n = 0, 1, 2, 3, 4$:

$$h_0\,h_1\,h_2\,h_3\,h_4 = a\,\bar{c}\,b\,a\,c\,\bar{b}\,a\,\bar{c}\,b\,\bar{a}\,c\,b\,a\,\bar{c}\,b \qquad \text{where, } h_0 = \varepsilon.$$

For an infinite number of disks ($n = 1, 2, 3, \ldots$), there will be a sequence H of *infinite* length:

$$H = h_0\,h_1\,h_2\,h_3 \ldots = a\,\bar{c}\,b\,a\,c\,\bar{b}\,a \ldots$$

This is an important result which can be used for the purpose of designing a one-time pad cryptosystem.

Definition:

We say a finite string y is a factor of a (finite or infinite) string $u = xyz$, where x is a finite string. We say a (finite or infinite) string u is squarefree (or repeat-free) if it has no factors of the form yy, where y is a finite, nonempty string. While it is easy to see that no string of length 4 or more over an alphabet or two letters can be squarefree, Thue produced an infinite squarefree string s over a three-letter alphabet; see [3].

Allouche, Astoorian, Randall and Shallit [3] presented the following theorem:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Theorem:

The Towers of Hanoi sequence H contains no factor of the form xx, for x a nonempty string [3].

See [3] for the proof.

Squarefree-Based One-Time Pad Cryptosystem

The strength of a one-time pad cryptosystem is that a large set of random key letters form a pad.

The squarefree sequence *H* can be as large as we wish for values of *n* and presumably a good candidate for a one-time pad cryptosystem. However, looking at this sequence closely reveals a weakness, a quasi-periodicity occurring every six positions. There are several ways around this problem. One solution would alternate among complementing, reversing or both complementing and reversing the repeated bits.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX

In this section we will present an implementation of the Padluc algorithm using the C programming language. Both Kasiski and Chi-square tests performed on the Padluc algorithm are included in this implementation.

A.1 The Padluc Algorithm -- C Implementation

```
/****************************************
 * The Padluc Program
 ****************************************/


/****************************************
 * Includes
 ****************************************/
# include <stdio.h>          /* i/o routines          */
# include <math.h>           /* pow()                 */
# include <stdlib.h>         /* malloc()              */
# include <string.h>         /* strlen(), strcat()    */
# include <time.h>           /* timing functions      */
/****************************************
 * Utility functions
 ****************************************/

/* Pause before continuing */

int pause()
{
    printf("Press <ENTER> to continue.\n");
    getchar();
    return 1;
}

/* Catastrophic failure */

int die(const char *msg)
{
    fprintf(stderr, "ERROR: %s\n", strerror(errno));
    fprintf(stderr, "MESSAGE: %s\n", msg);
    pause();
    exit(1);
    return 0;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/****************************************
 * StringToBin and BinToString
 ****************************************/
/*
 * Although it is much less efficient, in order to
 * simplify the algorithm and avoid having to worry
 * about byte boundaries, we represent binary
 * strings of data as null-terminated char*, with each character
 * in ('0','1') being one bit. These utility functions
 * translate a typical ASCII string to and from this representation.
 * Note that the bin representation we use here will be exactly
 * eight times as long as the ASCII representation.
 */

/*
 * Convert an ASCII string to a binary string. Assume that
 * result has enough space allocated to hold the output,
 * i.e. eight times strlen(ascii).
 */

int asciiToBin(const char *ascii, char *result)
{

    /* count */

    int i;

    /* Mask (to get bit values) */

    int mask[8] = {1,2,4,8,16,32,64,128};

    /* Walk the ascii string */

    for (i=0; i< (int) strlen(ascii); ++i)
    {
            /* The character we're looking at */

            int c = ascii[i];

            /* Bit counter */

            int b;

            /* Walk the bits */

            for (b=0; b<8; ++b)
            {
                    /* Build the bin representation bit by bit */
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                        if (c & mask[b])
                                result[8*i+b] = '1';
                        else
                                result[8*i+b] = '0';
                }
        }

        /* Null terminate the bin representation */

        result[8 * strlen(ascii)] = 0;

        /* Return success */

        return 1;
}

/*
 * Convert a binary string to ASCII.  Assume that
 * result has enough space allocated to hold the output,
 * i.e. 1/8 of strlen(bin).
 */

int binToAscii(const char *bin, char *result)
{

        /* count */

        int i;

        /* Mask for bits */

        int mask[8] = {1,2,4,8,16,32,64,128};

        /* Walk the bin representation (8 characters at a time)*/

        for (i=0; i< (int) strlen(bin) / 8; ++i)
        {
                /* Bit counter */

                int b;

                /* Our character (start at 0)*/

                result[i] = 0;

                /* Walk the bits, accumulating the character */

                for (b=0; b<8; ++b)
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                if (bin[8*i+b] == '1')
                        result[i] += mask[b];
        }

        /* Null-terminate the ASCII representation */

        result[strlen(bin) / 8] = 0;


        return 1;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/****************************************
 * Get the plaintext
 ****************************************/
/*
 * Allocate space and read a plain text from the file "plain_text.txt"
 */

char *getPlainAscii()
{
    /* The length of the file */

    int len;

    /* Pointer to the buffer */

    static char *plain_text = 0;

    /* File pointer */

    FILE *f = fopen("plain_text.txt", "r");

    /*
     * We're returning a pointer to a static buffer. If this function is called
     * a second time, free the memore and reload.
     */

    if (plain_text)
            free(plain_text);

    /* If we can't open the file, die. */

    if (!f)
            die("Unable to open file plain_text.txt");

    /*
     * To determine the length of the file, seek to the end, check our position,
     * and seek back to the beginning.  If any of these fails, die.
     */

    if (fseek(f, 0, SEEK_END))
            die("Unable to seek to end of file.");

    if (fgetpos(f, &len))
            die("Unable to get length of file.");

    if (fseek(f, 0, SEEK_SET))
            die("Unable to seek to beginning of file.");
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    /* Allocate enough space or die trying */

    plain_text = (char *) malloc(len + 1);
    if (!plain_text)
            die("Unable to allocate memory for file plain_text.txt.");

    /* Read the data */

    if (!fread(plain_text, 1, len, f))
            die("Unable to read entire file plain_text.txt.");


    fclose(f);

    /* Return the plaintext */

    return plain_text;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/****************************************
 * Get sequence for a given n, c
 ****************************************/
/*
 * This implements a portion of the Towers of Hanoi problem, finding
 * the state (recursively) for a given number of disks and number of steps.
 *
 * PSEUDOCODE:
 * PROCEDURE getSequence(configuration_num, num_disks, from_tower, to_tower,
spare_tower)
 *  IF (n = 0)
 *     do nothing
 *  IF (c < 2^(n-1))
 *     OUTPUT from_tower
 *     getSequence(configuration_num, n-1, from_tower, spare_tower, to_tower)
 *  ELSE
 *     OUTPUT to_tower
 *     getSequence(configuration_num - 2^(n-1), n-1, spare_tower, to_tower, from_tower)
 */

int getSequence(long int c, long int n,
                    const char *from_tower, const char *to_tower, const char
*spare_tower,
                    char *result)
{
    if (!n)
    {
            /* Base case -- do nothing */

            return 1;
    }

    if (c < (long int) pow(2, n-1)) /* First half */
    {
            /* Big disk is on from_tower */

            strcat(result, from_tower);

            /* Move remaining disks from from_tower to spare_tower */

            getSequence(c, n-1, from_tower, spare_tower, to_tower, result);
    }
    else /* c >= pow(2, n-1) -- second half */
    {
            /* Big disk on to_tower */

            strcat(result, to_tower);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            /* Move remaining disks from spare_tower to to_tower */

            getSequence(c - (long int) pow(2, n-1), n-1, spare_tower, to_tower, from_tower,
result);
    }

    return 1;
}

/****************************************
 * XOR on char *
 ****************************************/
/*
 * Assume a and b are strings containing only the characters
 * '1' and '0'.  The string result will be set to the
 * xor of a and b.  result will be as long as the minimum
 * length of a and b.  result is assumed to have enough space
 * allocated to hold this.
 */

int myXor(const char *a, const char *b, char *result)
{

    /* Count */

    int i;

    /* Our result will have a length equal to the min of lengths of a and b. */

    int len = ( strlen(a) > strlen(b) ? strlen(b) : strlen(a) );

    /* Walk the strings */

    for (i=0; i<len; ++i)
    {
            /*
             * Compute XOR on characters '0' and '1'
             */

            if ((a[i] == '1' && b[i] == '0') ||
                    (a[i] == '0' && b[i] == '1'))
                    result[i] = '1';
            else
                    result[i] = '0';
    }

    /* Null-terminate the result */

    result[len] = 0;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    return 1;
}


/****************************************
 * Huffman Encode
 ****************************************/
/*
 * This is a special Huffman encoding. The alphabet is
 * exactly three letters, with representations "0", "10", and "11".
 */

int myHuffman(const char *data,
              const char *tower_0,
              const char *tower_1,
              const char *tower_2,
              char *result)
{
    /* Get freqeuencies */

    int freq[3] = {0,0,0};

    /* The three outputs */

    char code[3][3];

    /* Get the "letter" length and data length */

    int letter_len = strlen(tower_0);
    int data_len = strlen(data);

    /* counter */

    int i;

    /* buffer */

    char *buf = (char *) malloc(letter_len + 2);

    /* Calculate frequencies */

    for (i=0; i<data_len; i += letter_len)
    {
        /* Copy the current letter to buf */

        strncpy(buf, data+i, letter_len);
        buf[letter_len] = 0;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
		/* Which letter is it?  Increment that frequency */

		if (!strcmp(buf, tower_0))
			++freq[0];
		else if (!strcmp(buf, tower_1))
			++freq[1];
		else if (!strcmp(buf, tower_2))
			++freq[2];
		else
			die("Bad string passed to Huffman.");
	}

	/* Make the code -- hardcode the six possibilities */

	if (freq[0] >= freq[1] && freq[1] >= freq[2])
	{ strcpy(code[0], "1"); strcpy(code[1], "01"); strcpy(code[2], "00"); }
	else if (freq[0] >= freq[2] && freq[2] >= freq[1])
	{ strcpy(code[0], "1"); strcpy(code[2], "01"); strcpy(code[1], "00"); }
	else if (freq[1] >= freq[0] && freq[0] >= freq[2])
	{ strcpy(code[1], "1"); strcpy(code[0], "01"); strcpy(code[2], "00"); }
	else if (freq[1] >= freq[2] && freq[2] >= freq[0])
	{ strcpy(code[1], "1"); strcpy(code[2], "01"); strcpy(code[0], "00"); }
	else if (freq[2] >= freq[1] && freq[1] >= freq[0])
	{ strcpy(code[2], "1"); strcpy(code[1], "01"); strcpy(code[0], "00"); }
	else if (freq[2] >= freq[0] && freq[0] >= freq[1])
	{ strcpy(code[2], "1"); strcpy(code[0], "01"); strcpy(code[1], "00"); }

	/* Compute the result */

	for (i=0; i<data_len; i += letter_len)
	{
		/* Again, copy the letter to buf */

		strncpy(buf, data+i, letter_len);
		buf[letter_len] = 0;

		/* This time, concatenate the appropriate code into result */

		if (!strcmp(buf, tower_0))
			strcat(result, code[0]);
		else if (!strcmp(buf, tower_1))
			strcat(result, code[1]);
		else if (!strcmp(buf, tower_2))
			strcat(result, code[2]);
		else
			die("Bad string passed to Huffman.");
	}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    /* Free the allocated buffer */

    free(buf);

    /* All done now */

    return 1;
}


/****************************************
 * Kasiski
 ****************************************/
/*
 * Search a string for surprisingly close repetitions of substrings.
 */

/*
 * Utility function. Find the distance to substring chunk in data.
 * data_len=strlen(data) and chunk_len=strlen(chunk) are precomputed and
 * passed in, since this function is called many times. Return -1 if
 * chunk does not appear in data.
 */

long int kasiskiSearch(char *data, int data_len, char *chunk, int chunk_len)
{
    /* Position count */

    int i;

    /* Walk the string, looking for chunk. Must be byte-aligned. */

    for (i=0; i<data_len - chunk_len; i += 8)
        if (!strncmp(data+i, chunk, chunk_len))
            return i;

    /* Failure */

    return -1;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/*
 * Our Kasiski test.  Search only byte-aligned.
 * -- data is the data to run Kasiski.
 * -- chunk_len is the length of substrings to search (should be multiple of 8)
 * -- alpha is fraction of expected distance for reporting.
 */

int kasiski(char *data, int chunk_len, double alpha)
{
    /* Count */

    int i;

    /* Allocate space for the chunk */

    char *chunk = (char *) malloc(chunk_len + 1);

    /* Compute the data length */

    int data_len = strlen(data);

    /* Compute the expected distance (2^chunk_len) */

    long int expected_distance = (long int) pow(2, chunk_len);

    /* Walk the data byte by byte */

    for (i=0; i<data_len-chunk_len; i+=8)
    {
            /* found distance */

            long int distance;

            /* Build the current chunk */

            strncpy(chunk, data+i, chunk_len);
            chunk[chunk_len] = 0;

            /* Find the distance */

            distance = kasiskiSearch( data + i + chunk_len , data_len - i - chunk_len,
                                                        chunk, chunk_len);

            /*
             * If we found the chunk, AND
             * it's surprisingly close, output a line of data
             */
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            if (distance != -1 &&
                    (double) distance / (double) expected_distance < alpha )
                    printf("%d\t%d\t%ld\t%s\n", chunk_len/8, i/8, distance/8, chunk);
    }

    /* Free allocated chunk */

    free(chunk);

    return 1;
}


/****************************************
 * Chi-square
 ****************************************/
/*
 * Run a chi-square test on 8-bit chunks
 * of data against the hypothesis that each byte
 * should appear 1/256th of the times.
 * count frequency of each of the 256 possible bytes
 */

int chi(const char *data)
{
    /* Count of each character */

    double count[256];

    /* Counters */

    int i,j;

    /* Data length */

    int data_len = strlen(data);

    /* n = data_len/8 */

    double n = (double) data_len / (double) 8;

    /* Expected frequency of each letter */

    double expected = n / (double) 256;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        /* Precompute some power */

        int pows[8] =
        { 1,          2,          4,          8,
          16,         32,         64,         128 };

        /* The chi-square statistic */

        double V = 0;

        /* Initialize counts to zero */

        for (i=0; i<256; ++i)
                count[i] = 0;


        /* Walk the data by bytes */

        for (i=0; i<data_len; i+=8)
        {
                /* Figure out which byte to increment */

                int pos=0;
                for (j=0; j<8; ++j)
                        if (data[i+j] == '1')
                                pos += pows[j];
                count[pos] += 1;
        }

        /* Output our counts */

        printf("i\tcount\n");
        for (i=0; i<256; ++i)
                printf("%d\t%f\n", i, count[i]);

        /* Compute the chi-square statistic */

        for (i=0; i<256; ++i)
                V += (count[i] - expected) * (count[i] - expected) / expected;

        /* Output the results */

        printf("n=%f V=%f\n", n, V);

        return 1;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/****************************************
 ****************************************
 * Main Program
 ****************************************
 ****************************************/

int main(char **argv, char **envp)
{
    /* Representations for the three towers (private key) */

    const char *from_tower = "1010";
    const char *to_tower = "1100";
    const char *spare_tower = "0011";

    /* Primes p and q (private key) */

    long int p = 31121;
    long int q = 31183;

    /*
     * Starting value for x.  (private key)
     * We start x at (p+q) / 4, to ensure that x^2 will be greater than
     * either p or q (i.e. a_p and a_q will probably be different immediately.)
     */

    long int start_x = (p + q) / 4;

    /* Number of disks */

    long int n = 15;

    /* Plaintext */

    char *plain_ascii = getPlainAscii();

    /*
     * Plain binary (only characters '1' and '0').  We precompute the length
     * for efficiency, but this is still a null-terminated string
     */

    char *plain_bin;
    long int plain_bin_len = 0;

    /* Key buffer.  Again we keep track of length for efficiency. */

    char *Padluc_key;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/* Ciphertext */

char *cipher_bin;

/* Sequence buffers. */

char *buf_p;            /* Buffer for prime p */
char *buf_p_h;          /* Huffman encoded buffer for prime p */
char *buf_q;            /* Buffer for prime q */
char *buf_q_h;          /* Huffman encoded buffer for prime q */
char *buf;                      /* General buffer (i.e. buf_p XOR buf_q */
char *buf_h;            /* Huffman encoded buf */

/*
 * Iterate using x.  We take a_p and a_q to be quadratic residues of x mod p and q,
 * respectively.
 */

long int x = start_x;
long int a_p = 1;
long int a_q = 1;

/* Timing variables */

clock_t start_time = 0;
clock_t finish_time = 0;

/* Counters */

int j;



/* Translate the plainAscii into plainBin, the binary represenation */

plain_bin = (char *) malloc(8 * (strlen(plain_ascii) + 1));
asciiToBin(plain_ascii, plain_bin);
plain_bin_len = strlen(plain_bin);

/*
 * Allocate enough space for the key.  The key may be as much as
 * n*tower_rep_len longer than the plaintext.  Add a little extra for safety.
 * Initialize to the empty string.  Allocate similar space for the ciphertext
 */

Padluc_key = (char *) malloc(strlen(plain_bin) + 2 * n * strlen(from_tower));
Padluc_key[0] = 0;
cipher_bin = (char *) malloc(strlen(plain_bin) + 2 * n * strlen(from_tower));
cipher_bin[0] = 0;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/*
 * Allocate enough space for the sequence buffers.  They require 2*n*tower_rep_len
 * characters, plus one for null-termination
 */

buf_p   = (char *) malloc(n * strlen(from_tower) + 1);
buf_p_h = (char *) malloc(n * strlen(from_tower) + 1);
buf_q   = (char *) malloc(n * strlen(from_tower) + 1);
buf_q_h = (char *) malloc(n * strlen(from_tower) + 1);
buf     = (char *) malloc(n * strlen(from_tower) + 1);
buf_h   = (char *) malloc(n * strlen(from_tower) + 1);

/* Start timing now */

start_time = clock();

/*
 * Main algorithm:  while we need more key... (note that the various
 * keys might be of different lengths, since Huffman changes the length)
 */

while ( (long int) strlen(Padluc_key) < plain_bin_len )
{
        /* Initialize buffers */

        buf_p[0] = 0;
        buf_p_h[0] = 0;
        buf_q[0] = 0;
        buf_q_h[0] = 0;
        buf[0] = 0;

        /* Increment x */

        x++;

        /* Make sure that x hasn't cycled */

        if (x > (long int) pow(2, n) - 1)
                die("n too small for current plaintext.");

        /* Compute quadratic residues for p and q */

        a_p = (long int) x*x % (long int) p;
        a_q = (long int) x*x % (long int) q;
        if (a_p < 0 || a_q < 0)
                die("Overflow.");

        /* Compute the sequences a_p and a_q */
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            getSequence(a_p, n, from_tower, to_tower, spare_tower, buf_p);
            getSequence(a_q, n, from_tower, to_tower, spare_tower, buf_q);

            /* Huffman encode the sequences */

            myHuffman(buf_p, from_tower, to_tower, spare_tower, buf_p_h);
            myHuffman(buf_q, from_tower, to_tower, spare_tower, buf_q_h);

            /* XOR the sequences */

            myXor(buf_p, buf_q, buf);
            myXor(buf_p_h, buf_q_h, buf_h);

            /* Concatenate the result. */

            strcat(Padluc_key, buf_h);
    }

    /* XOR with the plaintext to get the ciphertext */

    myXor(Padluc_key, plain_bin, cipher_bin);

    /* Finish timing now and output */

    finish_time = clock();
    /* printf("Time: %0.4f\n",
                        ( (double) finish_time - (double) start_time) / (double)
CLOCKS_PER_SEC); */

    /* Output the various results */

    printf("Plaintext:\n%s\n\nKey:\n%s\n\nCiphertext:\n%s\n\n",
            plain_bin, Padluc_key, cipher_bin);

    /* Output a chi-square test on one-byte chunks for each. */

    printf("Chi-square test for cipher_bin\n");
    chi(cipher_bin);

    /*
     * Kasiski search each cipher_bin for suprising results.
     * Define "surprising" as 1/100 as close together as expected.
     * Only go up to 128-bit chunks -- if we find any of those, we're
     * pretty sure something's wrong.
     */

    printf("Kasiski for cipher_bin\n");
    printf("len\tpos\tdist\tchunk\n");
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    for (j=16; j<128; j+=8)
        kasiski(cipher_bin, j, 0.01);

    /* Be good!  Free all dynamically allocated space. */

    free(Padluc_key);
    free(cipher_bin);
    free(plain_ascii);
    free(plain_bin);
    free(buf_p);
    free(buf_p_h);
    free(buf_q);
    free(buf_q_h);
    free(buf);
    free(buf_h);

    /* Completed */

    return 0;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A.2 Sample Plaintext

INTRODUCTION

1. The Industrial Revolution and its consequences have been a disaster for the human race. They have greatly increased the life-expectancy of those of us who live in "advanced" countries, but they have destabilized society, have made life unfulfilling, have subjected human beings to indignities, have led to widespread psychological suffering (in the Third World to physical suffering as well) and have inflicted severe damage on the natural world. The continued development of technology will worsen the situation. It will certainly subject human beings to greater indignities and inflict greater damage on the natural world, it will probably lead to greater social disruption and psychological suffering, and it may lead to increased physical suffering even in "advanced" countries.

2. The industrial-technological system may survive or it may break down. If it survives, it MAY eventually achieve a low level of physical and psychological suffering, but only after passing through a long and very painful period of adjustment and only at the cost of permanently reducing human beings and many other living organisms to engineered products and mere cogs in the social machine. Furthermore, if the system survives, the consequences will be inevitable: There is no way of reforming or modifying the system so as to prevent it from depriving people of dignity and autonomy.

3. If the system breaks down the consequences will still be very painful. But the bigger the system grows the more disastrous the results of its breakdown will be, so if it is to break down it had best break down sooner rather than later.

4. We therefore advocate a revolution against the industrial system. This revolution may or may not make use of violence: it may be sudden or it may be a relatively gradual process spanning a few decades. We can't predict any of that. But we do outline in a very general way the measures that those who hate the industrial system should take in order to prepare the way for a revolution against that form of society. This is not to be a POLITICAL revolution. Its object will be to overthrow not governments but the economic and technological basis of the present society.

5. In this article we give attention to only some of the negative developments that have grown out of the industrial-technological system. Other such developments we mention only briefly or ignore altogether. This does not mean that we regard these other developments as unimportant. For practical reasons we have to confine our discussion to areas that have received insufficient public attention or in which we have something new to say. For example, since there are well-developed environmental and wilderness movements, we have written very little about environmental degradation or the destruction of wild nature, even though we consider these to be highly important.

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

xxxxx

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A.3 Sample Output of the Padluc Implementation

Plaintext:

```
0000010000000100100100100111001000101010010010101111001000100010101010
1100001000101010100100101111001001110010010100000000010000000100010100000
0000010000000010000000100010100000000010000000100000001000101000000000100
0000010000000010010001100011101000000010000101010000101101010011000000100
1001001001110110001001101010111011001110001011100100111010010110100000110
0011011000000100010010101010011001101110111101100011011010101110001011110
1001011011110110011101100000010010000110011101100010011000000100100101110
0010111011001110000000100110001101111011001110110110011101010011010001110
1010111010100110011101101100011010100110110011100000001000001011010000110
0110111010100110000000100010001101010011010100110011101100000001001000110
0000010000100110100101101100111010000110110011100010111010100110010011110
0101000000000100000001000000010001100110111101100100111000000100000101110
0001011010100110000000100000101101010111010110110100001100111011000000100
0100111010000110110001101010011001110100000000100001010100001011010100110
1001111000000100000101101000011001101110101001100000010011100110010011110
1010011010000110001011100011011010011110000000100100101100111011011000110
0100111010100110100001101100111010100110001001100000010000101110000101110
1010011000000100001101101001011001100110101001101011010010100110000111110
0000111010100110110001100010111010000110011101101100011010011110000000100
1111011001100110010100000000010000000100000001000010111000010110111110110
1100111010100110000000100111101100110011000000100101011101100111000000100
1110111000010110111101100000001000011011010010110011011101010011000000100
1001011001110110000000100010001001000011000100110011011101000011001110110
1100011010100110001001100100010000000100110001101111011010101110011101110
0010111001001110100101101010011011001110001101000000010001000110101011110
0010111000000100001011100001011010100110100111100000001000001011010000110
0110111010100110010100000000010000000100000001000010011010100110110011110
0010111010000110010001101001011000110110100101100101111010100110001001110
0000010011001110111101101100011010010110101001100010111010011110001101000
0000010000010110100001100110111010100110000000100101101101000011000100110
1010011000000100001101101001011001100110101001100000010010101110011101110
0110011010101110001101100110011010010110001101100011011010010110011101110
1110011000110100000000100000101101000011001101110101001100000010011001110
1010111001000110010101101010011011000110001011101010011000100110010100000
0000010000000010000000100000101101010111010110110100001100111011000000100
0100011010100110100101100111011011100110110011100000001000010111011110110
0000010010010110011101100010011010010110111001100111011010010110001011110
1001011010100110110011100011010000000010000010110100001100110111010100110
0000010000110110101001100010011000000010000101110111101100000001001110110
1001011000100110101001101100111000001110010011101010011010000110001001110
0000010000000111011001110100111101100011000010110111101100011011011110110
1110011010010110110001101000011000110110010100000000010000000100000000100
1100111010101110011001100110011010100110010011101001011001110110111001110
0000010000010100100101100111011000000010000101110000101101010011000000100
0010101000010110100101100100111000100110000000100111010101111011001001110
0011011000100110000000100000101110111101100000001000000111000010110100111110
1100111010010110110001101000011000110110000000100110011101010111001100110
0110011010100110010011101001011001110110111001100000001001000011011001110
0000010011101110101001100011011000110110100101000000010010000110011101110
0010011000000100000010110100001100110111010100110010100000000010000000100
0000010010010110011101100110011000110110100101101100011000101110101001110
0010011000000100110011101010011001101110101001100100111010100110000000100
0010011010000110101101101000011011100110101001100000010011110110011101110
0000010000101110000101101010011000000010001110110100001100010111010101110
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

01001110100001100011011000000100111011101111011001001110001101100010011 0

Key:

```
0001000100100100111001101010101110100110100011111110101001000011000010 11
1111100000100100010001111110101001111001101111011110100101111111111111101
1001101100001000111011110110001010000101110111101001000011101011101010 01
1110100011111101110000000101001000000000110110000111010111100101001010 00
1000001000011001011011010101111110000010000010000111101001101001000000 11
0100001001100001000010011111000110100000010101100010000000001011101101 01
0000000010011011100001010000100100010011011000000001100101000110100000 00
0000000000001011010001010001111001010100100010100011110110010110110100 01
0111111111001000101010101000110000000000110111001010111000011111011111 01
1111000011111011000011110001011010101010101011000000111110101011111110 00
0111110010101101101101100111000000101010100001011000101111000100101111 10
1111111011001100010011010000011110011000111000011111000100011000000000 00
0101010111010111110011101010110010101011111110001000100101111111101110 01
0000111011000101100110001001011100110111100000100001001101111111000000 110
1001011001001110000000100010001001100110010001010111110110011100000101 101
0110111001111011100111011110001110110100110111000110100101010100101000 11
0100100001010111100000011111111101011001000111100011001110111001100111 10
0011110101000010111011101101101010101111100000110101111100010000011011 10
1101011111111001010111100100000011010010111011111111110011100000000110 100
0011110000000000000110011110001010001010111110100000001111111011100000 11
0100011001111001001111000110100010110010000100011111001000101111101000 11
1100101110111100101111110000101010011011110011110000011111101101011011 11
0100101010110100110100001100000101101111000010011000101100001110011110 10
1010111001000010000001011101000010110000000011111100101111010010101001 11
1100101011010111101101010000000101111010010010101010100000000000101101 110
1111101101011111110011101000111001001001011110010011011000000010101101 10
0101001100001001101000110111110110101011010100010101101101011111000100 01
1101111111110000011010000100001100010011001000000101011111100001010111 100
1011010000011101000110010000010100101100010111110000111111110001001000 00
1001001100001001010101010100010101000001111001111110011100011101101100 01
1111011111100010010100111111000110000000101011010111011111101111001010 10
0001111101001001001101001110101100110001010101100001011000100110011111 10
0000011010111001110001111100110011011000011101000001010101010100011100 11
0001010110000100010001111111010111111110101000010001100100010010001101 01
1111011110010000010110111011010001101001100100000100000000101000110100 101
1001011101000101100111011011111101101000001100111011000011000101110111 10
0011011011011001101111110010111000000001000000100100110000110000100101 111
0000100100001000101110011101000111100110000110000111000111110101001010 00
0100011001000000111010000101100000000000000000110100111011101011111100 000
0100001000101010110100001011000011101101101000101100011001101000111011 00
1011010010000000001111011001000000000111110010101111111011010010010110 100
1000010010001110000000110010111111111110101011100011111110010111000001 1110
0000100010011001001010111101001000111101011110101011111010001001111111 00
1110010001111001011000001010111110011001010011001010000010000110010001 10
1111000111101010011101111010010101001110000011101010000011110010011000 00
0000000111111000010110010110110110110000101101100100001001001011001111 01
1011011011000000000100110011000000101101011110001011000110100001001100 11
0000100111011011000011010111101111101111000001110110111011010011111011 11
0101100001010111000111001111001010011100101001010100000110010111011101 11
1010011100111101001011011101101000111111111011011111011110001100110001 00
1100011011000000111110000010001110110110000101011101101101010110101011 00
1000000010001110110100101100010100101111101001101001110101010101011100 10
1011011010101001001101110110011001011000100001100000100110010001001011 11
1101000000101011110010100110110010111100101011101110111000000100100011 11
1111000000001000000100110111111011010001010011110100100000001010110100 011001
0100010101111011100001111100010001101100110011100001101111110000100010 100
0010111010010101101001100010100011111111111001110010001010111000100101 01
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1111101111000000101101001101010000000001011000100110010000000001010010011 0
011110111001000010101101110101011000101011011000101001101010101000110111
001111101000110001110011010010000000101100100100110110111101001010101101
101001110001010110001000100011110100011010000101011010000101111101010101
110000110100100011101101111111011101100111100001011001011011001101010101
011100011000110110111110000111110001111010100011110100101101111110001010
000001011001011110011110101111011101111011010010101111111111001011011110
011010111011010110011110101111011110100000101001111100011110000100100100
110000000101001001011100101000000011111100001111100010100011011100001100
001001010100000000001010010001110011010011000010101101010100111000011111
101110011101111111100001011111011011110111101011011011100011010110110011
110000101111011101111001111110100100011110011101111111101111111101111111
10000100010010100000000011001100110110000000101001010100110110100000101000
110111111100101000110100011101000101100010101101010001110110100011011111
1101111111010111011110100011110100000001000001101111111001000111111111100 00
0000010001111010011010100000000000001010011110000110000000001010111010100
001001000001001111010110001100010100001011011010001000000110100001111111
100111010010010000000000111010101011101011001001011011011010111101011000
01000010001011000000000101001100111010000000000000010011000010000000000010
110111010000000000110000000001010110100101011010111000001010101101001101
0000110111101111110010110100101000100001010000100011010000110100110111100
001010010011100100101011000100010111111000010000110011111100100010101110
1101111111000101111100100100111100010000000011111010100000000001111010111
100001001000111101001000010100010111100100011110010100010101101011111111
0000011111100110100111111111000101111101001000101110000001111101001010100
0111110011100011111101001010000000000000111001000001010001011011111100010
111100111111011011000001101000000010111001101000010101111011111111100111
000011010001001101101011010001000111010011101101011110101010100000010011
101000000111001010101000110111111010110110110100100101110011010110111000
000000100001101110111011010010000001101011110010000101101110100111110100
101101100111011101111010010111100110100111011000011110011010001000000011
010000100011011001111000111101110010010111111001110111010000000110110100
000010110111010100110011110111000010011110010011010000110001011010111011
111111100011110110111100011111100000011100101001010100110110100010101110
111001010101101100110110001000100010010000000101110010100010100101010001
001001100010101001000000101101011100100010010101001010110010110000001111
011100010010100000101101100100110111110111100100101000011001000110111010
000000000001111101010001000011001010101100110110000010010011101100001111
1100010010101011101011001000000000001100011100111011010100101010001000011
111111100111111010011111010101001111001111111111000000101111110000000110011
111110011001000111001011111010011110111000000001101100000101001111110110
101101111011000101010101111000011011001111100010010010110111011110100100
0110011010001010011000001111100001001110111111100011111001010110101011001
000000001110110101110000110100010111011100101101101101000010100100000101
110111011101101000100010001110010101111100100111000110110110000001010000
100110011100111110101001010111111011100100110010000110011011010101011010
1011011011111101100101100101010010011000001010111101001100000100111100100
100100001100001110100101001101000001100011001100000101011110110011101110
100011011111000111000010000011010011111110110101110100011011110010101001
011010101111001010100100011001011011010001011001001010100111010001001000
1000011101011000101111000011000100111000000000001100010010101101000101110
0101110110110110000110001110110000010100101110000001011101100010001000000
110011001011110011100100110001111111101111101001001011110100110110000101
0101001001111010010111011100010000001010000011000100110000001001000100010
010111111101001111111111101110101101111111010100110010111100000110101010
010001101011001100010101101011100110100101111101010110011100100111001111
110010000010101100101110011011001010010101001100010011100111110011011001
010101110110000101100110100010100011011010101011101000011101001110100000
011000001111101111110001001110001100111111010011111100110101110010110000
0001100001010110100011101001000010110010000001111110111111110111110001111

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Ciphertext:

```
000101010010000000111010011011001100011001100010100011000011000011010000001
001110100000111011010101000110000000101111101101111011010111101110101101
100111110000110011101011001100101000000111011010100101001011101110101101
111011001111100101001100001001100000010011110010011000110100001100101100
000100000011011110100101111111000101001100001001100011010011111111110000101
011101000110010101000011010101111100111010100000000101101010010110011011
100101100110110111110011000011011001010100010110001111110100001000010110
001011101100010101000000111011000101000101111110011110011001100000101111
110100010110111011011100010010101010011000010010101010100000100111111011
100111100101110100001011010100000000011000000101001111001101011110111110
011110001000101100100000010111110101011000100101110100101011000101111000
101011101100100001001001000000111111111000010111101111110001110000101110
010000110111000111001010101110100000010101001110000011110000100110111101
010000000100001101011110001100010100001110000110001110010110100010100000
000010000100101000010010110000101010001000101100111111111101111001100011
110010001111110110110011110101010010101011011000111111100100010011100101
000001101111000100000111001100011111111100111000000110111100101111000100
100110110100011011011000010011001100100100100101111010111011011001110000
110110010101111111001100001101110010101001001100100111010011111100011000
110010100110011001001001111001101000111011111110001011011110110101110101
100010001101111100111000100111101101010000010101010111001110000110100111
001001011010101001001001000011101010110101011001011010010100101101101011
110111001100001011010100100001011110100100101111111001011000100000001100
011010011000011100100011100101001011010011001001001111010111111001101000
111001001001100100100011101001000011101010100001010101000100011111000000
110101010101101111110000010011000111011111111001110011001000010100001100
001111011010111111110011011110011010111101010101011111011111100111011111
111100010110011010010110000100000001000011010111000000010010001101011010
101100001101001111101111111000011101110101111100100100001011011110001010
100101110001111111010011001010111110011111100011010100011001101110010111
010100011110011001100101011001111110011000001011011100110100000101011100
011110011110011100000010100011011010011101100000000100000101100000000100
111000001000110111000011110110100101111000011010101100110101000010111101
101110111100001000010001010100110011100010001111101111110011010001100101
111100111001010001011111101000101100011100100110110001100010011110100001
110100011110001100001011110010011000111011111101101101001110101100101000
001100100100111111001001000010001001010011100010111011101111011100000001
100111111010111001110111111001011110001000001110111101111001101110001110
010000100111011101110110100101100000010000100011110011011010101100101110
110101000000110001110110011111101110001111110110001100000111011101100101
101100001000111011110011000011101100100110000011000011010111111111001101
011000100001100011000000110110011100110011101000111110100101100000011010
110001100011011101001101101101001001101100110100001010001111111100011010
111000000110110111110110110110011001110101100010101101100010000001000010
110110111111110011100001111010110110100000000101001001010000000100001011
001101111110111100101110101000011010001101011001001001100010111011010001
011110000101011011010101101101100001101101111100011111110000111101010101
011011110111111010100001111101101100110011110000101101010010101010010000
010111001011100110111010110001001010101000110001010001010001000100000000
100000010011100100111011010111000101000101001011101001111000100011000000
110000100101011010001110010001011000000001000001100011101011110000000101
101001101000101000011100011000110100000010000000011010011111100110111011
100100000001011111100000001111000001011111100010000000000110101100111010
110101000000010111011100110010101011100011011000011010000010101000100000
101111101001011001111011111100100110010010001100000011101001101100111111
001100010111111110101101110100101100101011001010110111010001011101100010
000000000000001111010000100001100101100111000000101110010101111001001000
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

11111111111001100001001010111010101001001111001000111110000010111001000
11011101111001101000001111010001011111001011111010100010100001001001000
11111000100110100000010110111110001111011101001000111101010011001010100
01001001100000111011111010111001010000100110101110011110000100011001101
01100101001111101110100111010011110011110100011101100001011111011100001
01011111001000110011100000110001100010000101010110100100101010111000111
10010111101110011001101001010011010010001110010010001001111101100001100
11001101111110111011000000111011011111100101111111000111011111111011101
11000100010101100101100001101110100100010100100111011100100100011100101
00001011010001000001110011101001100000100100010011000011010010100101100
00011111010010011001011110011011011100111110111101000000110000111011011
00100100101110011101111101111100011010010011101110110000111110100110100
11110010011011001001011101111111111000110101111110111111011111111101000
00010001110011101011001000000001001111110101010011101000100011110101110
11101001010000011011110000010011000001101110101110110111101110010111011
00101010110111000010010000000100001100100111011001110110100100110011001
10000010010000111101001000110101010001100010110001010110011011000101000
10001011100000010000000100100111000011110011100111111000011111000011101
01110100001010001110110001101111100111100011011001101010000101000000011
01001011001011100011010011101011111111101101100110101101010111101000001
01000011000110011101000000001001000000100101100101111011001101101100010
10001111101111110000110100010101010100001110011011001011001011101110000
01111001010000111101110011101001010111100001101110011110111100010001000
00010010000001001011111110000000001011111010001101001010101011111000110
11001001101010000011000111111111010100111011010000010110100011000101000
11111010100101011101001010100100000011110000001101011011001110001111010
00000101110000000011011101000110110010100001011000101001010010011110001
11000011101111010000110100100010110100101010001111101100110111101111010
10010100011101100010111010101001100010111011000000000001000110111011110
10110100100111010010010101001100001011000101010010010000110011111111000
10011000100000010111111011001000000111110001111000110111000001001000010
10001100100100000101111010100111001000011111110111011001000011111010001
10010101101110111010010100011010101000011010010101000111110110000001010
10011000010110110001101000110000100100010101111110110101011011000000100
10001011111111010100000000100110101100100111001111001110011011011101011
00000000010001001100011011000011000011100011001100001101011010000000101
10110111110111101000001111100101010100111010101000110111001101110111010
01110100010011110101010100001000101011110110011000001101001111110000101
10001000110111111010100010101010000011100100000101101110110000100011010
11011000110100000101000101111010101111010110100010000011110001101000011
11010111001101110000110111111111001100011110111100001101010010100010000
00100000101110111110100111101011110110111001011001101010110111001100010
11000000001111000110010001001110110010000110001001111000011000111111011
01001110100000111110011010111111110100010010100101000010011001110000000
01001011111101000010011010001111110110011011100100011111001001100001111
00111111010010010111111110000111110111101001101100001110110010011101011
01011000100011010101100010101101101000100011000110100010100111111100101
10010100000011010000101101111010011101100101101001111011010010100010000
10111001111101010101010000100011001110110000011101010011001001101010110
11001100100111000000000100001001110011010111101111010110001000010011111
00011001010111000011101011110111001011101001011100101111001101001000100
01011001001100000001110011011010111000100101011000101010111100101110011
10100010000110101101001011000011000011011000111101111111010010011000000
01010110011101000100101101011010110001001001101010001010100101000001010
01011011001000011000100110011100110110111101101000000101010111110110110
01010000001000101001000110101100010001111111010111001111101001111111100
11001100111001011000000000001010110000111101010000000000111010101010111
10110001010101010110001011001100100110001000010110100101001001011101011
01010110011001011111010110111110101010011111110101010101000100101011010
00010110110100000100000000101111000100100011110010011100111101011101000

A.4 Chi-Square Test Result

The Chi-square test outputs the following:

- counts of each byte

- n = bytes in ciphertext

- V = Chi-square value

```
Chi-square test for cipher_bin
i     count
0     26.000000
1     28.000000
2     25.000000
3     24.000000
4     25.000000
5     25.000000
6     22.000000
7     41.000000
8     37.000000
9     29.000000
10    27.000000
11    29.000000
12    24.000000
13    29.000000
14    27.000000
15    15.000000
16    28.000000
17    29.000000
18    23.000000
19    22.000000
20    18.000000
21    20.000000
22    34.000000
23    27.000000
24    36.000000
25    30.000000
26    28.000000
27    28.000000
28    32.000000
29    16.000000
30    24.000000
31    22.000000
32    31.000000
33    36.000000
34    18.000000
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
35    20.000000
36    28.000000
37    32.000000
38    28.000000
39    20.000000
40    29.000000
41    25.000000
42    27.000000
43    21.000000
44    25.000000
45    24.000000
46    27.000000
47    25.000000
48    29.000000
49    31.000000
50    30.000000
51    20.000000
52    24.000000
53    27.000000
54    32.000000
55    27.000000
56    24.000000
57    16.000000
58    19.000000
59    29.000000
60    24.000000
61    29.000000
62    25.000000
63    23.000000
64    24.000000
65    20.000000
66    31.000000
67    29.000000
68    18.000000
69    26.000000
70    32.000000
71    30.000000
72    29.000000
73    24.000000
74    15.000000
75    23.000000
76    29.000000
77    19.000000
78    20.000000
79    24.000000
80    25.000000
81    22.000000
82    28.000000
83    41.000000
84    16.000000
85    18.000000
86    29.000000
87    34.000000
88    47.000000
89    36.000000
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

90	32.000000
91	25.000000
92	27.000000
93	26.000000
94	25.000000
95	32.000000
96	22.000000
97	19.000000
98	27.000000
99	20.000000
100	29.000000
101	24.000000
102	21.000000
103	31.000000
104	32.000000
105	26.000000
106	21.000000
107	23.000000
108	33.000000
109	27.000000
110	31.000000
111	26.000000
112	32.000000
113	25.000000
114	27.000000
115	30.000000
116	42.000000
117	24.000000
118	26.000000
119	21.000000
120	20.000000
121	34.000000
122	32.000000
123	26.000000
124	13.000000
125	35.000000
126	23.000000
127	24.000000
128	32.000000
129	31.000000
130	19.000000
131	15.000000
132	20.000000
133	23.000000
134	29.000000
135	24.000000
136	35.000000
137	36.000000
138	29.000000
139	22.000000
140	28.000000
141	30.000000
142	20.000000
143	21.000000
144	27.000000

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
145   14.000000
146   31.000000
147   31.000000
148   23.000000
149   27.000000
150   18.000000
151   27.000000
152   22.000000
153   17.000000
154   18.000000
155   26.000000
156   26.000000
157   33.000000
158   25.000000
159   19.000000
160   29.000000
161   32.000000
162   24.000000
163   25.000000
164   29.000000
165   33.000000
166   36.000000
167   29.000000
168   19.000000
169   27.000000
170   20.000000
171   24.000000
172   23.000000
173   19.000000
174   26.000000
175   17.000000
176   44.000000
177   27.000000
178   23.000000
179   18.000000
180   16.000000
181   29.000000
182   23.000000
183   25.000000
184   15.000000
185   26.000000
186   22.000000
187   18.000000
188   21.000000
189   36.000000
190   31.000000
191   20.000000
192   25.000000
193   27.000000
194   29.000000
195   27.000000
196   23.000000
197   22.000000
198   25.000000
199   30.000000
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
200   22.000000
201   17.000000
202   30.000000
203   18.000000
204   25.000000
205   19.000000
206   19.000000
207   30.000000
208   36.000000
209   25.000000
210   33.000000
211   25.000000
212   28.000000
213   22.000000
214   19.000000
215   17.000000
216   37.000000
217   31.000000
218   24.000000
219   21.000000
220   23.000000
221   23.000000
222   25.000000
223   29.000000
224   15.000000
225   25.000000
226   30.000000
227   15.000000
228   19.000000
229   28.000000
230   20.000000
231   41.000000
232   21.000000
233   19.000000
234   18.000000
235   23.000000
236   30.000000
237   27.000000
238   21.000000
239   26.000000
240   38.000000
241   25.000000
242   23.000000
243   25.000000
244   20.000000
245   23.000000
246   17.000000
247   17.000000
248   36.000000
249   27.000000
250   34.000000
251   15.000000
252   30.000000
253   39.000000
254   40.000000
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
255   27.000000

n=6601.000000 V=362.037116
```

The result demonstrates: $V = 332$, ~ 99% probability of a nonrandom distribution for 255 degrees of freedom.

A.5 Kasiski Test Result

Kasiski for cipher_bin

len	pos	dist		chunk
2	29	9	$= 3^2$	0100110000100110
2	2029	65	= 5.13	0010111000011100
2	2119	61	= 61	0111101101110100
2	2372	25	$= 5^2$	0100100001011100
2	3781	78	= 2.3.13	0111110000000011
2	3814	63	$= 3^2.7$	1010111100011000
2	4199	52	$= 2^2.13$	1110001000011000
2	5502	46	= 2.23	0000111010111100
2	5641	34	= 2.17	1111101101000100
3	1454	2838	= 2.3.11.43	010011100101111100011010

The result demonstrates: Only 10 "surprising" repetitions. All of these repetitions except one, are of length 2 bytes; the single exception is of length 3 bytes. There are no prominent divisors of distances.

A.6 The Padluc Algorithm and Three Variations

The C implementations of the Padluc algorithm and three alternative algorithms are shown. Each algorithm is shown to be subjected to Chi-square and Kasiski tests. In the final analysis, the test results are compared.

I. 1. Select 2 primes p and q, where $1 \leq p, q \leq 2^n - 1$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. Calculate quadratic residues for a given start_x using p and q

3. Call getSequence to find the sequences associated with the configuration numbers obtained from step 2

4. XOR the sequences obtained from step 3

5. Concatenate the result of step 3 with the content of key

6. If the key_buf_len ≥ plain_len

    XOR the plaintext and the key to get ciphertext

   else

    x = x + 1

    go to step 2

2\. 1. Select 2 primes p and q, where $1 \leq p \,\&\, q \leq 2^n - 1$

2. Calculate quadratic residue for a given start_x using p and q

3. Call getSequence to find the sequences associated configuration numbers obtained from step 2

4. Perform Huffman data compression on the sequences obtained from step 3

5. XOR the strings obtained from step 4

6. Concatenate the result of step 5 with the content of key

7. If the key_buf_len ≥ plain_len

    XOR the plaintext and the key to get ciphertext

   else

    x = x + 1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

go to step 2

3. Same as *2* except it's done on one prime

4. Same as *3* without Huffman encoding

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A.7 The Padluc Variations -- C Implementation

```
/****************************************
 * The Padluc Program
 ****************************************/

/****************************************
 * Includes
 ****************************************/
# include <stdio.h>         /* i/o routines        */
# include <math.h>          /* pow()               */
# include <stdlib.h>        /* malloc()            */
# include <string.h>        /* strlen(), strcat()  */
# include <time.h>          /* timing functions    */

/****************************************
 * Utility functions
 ****************************************/
/* Pause before continuing */
int pause()
{
      printf("Press <ENTER> to continue.\n");
      getchar();
      return 1;
}
/* Catastrophic failure */
int die(const char *msg)
{
      fprintf(stderr, "ERROR: %s\n", strerror(errno));
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    fprintf(stderr, "MESSAGE: %s\n", msg);
    pause();
    exit(1);
    return 0;
}

/****************************************
 * StringToBin and BinToString
 ****************************************/
/*
 * Although it is much less efficient, in order to
 * simplify the algorithm and avoid having to worry
 * about byte boundaries, we represent binary
 * strings of data as null-terminated char*, with each character
 * in ('0','1') being one bit.  These utility functions
 * translate a typical ASCII string to and from this representation.
 * Note that the bin representation we use here will be exactly
 * eight times as long as the ASCII representation.
 */

/*
 * Convert an ASCII string to a binary string.  Assume that
 * result has enough space allocated to hold the output,
 * i.e. eight times strlen(ascii).
 */

int asciiToBin(const char *ascii, char *result)
{
    /* Count */
    int i;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    /* Mask (to get bit values) */
    int mask[8] = {1,2,4,8,16,32,64,128};

    /* Walk the ascii string */
    for (i=0; i< (int) strlen(ascii); ++i)
    {
        /* The character we're looking at */
        int c = ascii[i];

        /* Bit counter */
        int b;

        /*Walk the bits */
        for (b=0; b<8; ++b)
        {
            /* Build the bin representation bit by bit */
            if (c & mask[b])
                result[8*i+b] = '1';
            else
                result[8*i÷b] = '0';
        }
    }
    /* Null terminate the bin representation */
    result[8 * strlen(ascii)] = 0;

    return 1;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/*
 * Convert a binary string to ASCII.  Assume that
 * result has enough space allocated to hold the output,
 * i.e. 1/8 of strlen(bin).
 */

int binToAscii(const char *bin, char *result)
{
    /* Counter */
    int i;

    /* Mask for bits */
    int mask[8] = {1,2,4,8,16,32,64,128};

    /* Walk the bin representation (8 characters at a time) */
    for (i=0; i< (int) strlen(bin) / 8; ++i)
    {
        /* Bit counter */
        int b;

        /* Our character (start at 0) */
        result[i] = 0;

        /* Walk the bits, accumulating the character */
        for (b=0; b<8; ++b)
            if (bin[8*i+b] == '1')
                result[i] += mask[b];
    }

    /* Null-terminate the ASCII representation */
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    result[strlen(bin) / 8] = 0;

     return 1;
}

/**************************************
 * Get the plaintext
 **************************************/
/* Allocate space and read a plain text from the file "plain_text.txt */
char *getPlainAscii()
{
    /* The length of the file */
    int len;

    /* Pointer to the buffer */
    static char *plain_text = 0;

    /* File pointer */
    FILE *f = fopen("plain_text.txt", "r");

    /*
     * We're returning a pointer to a static buffer. If this function is called
     * a second time, free the memore and reload.
     */
    if (plain_text)
        free(plain_text);
    /* If we can't open the file, die. */
    if (!f)
        die("Unable to open file plain_text.txt");
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    /*
     * To determine the length of the file, seek to the end, check our position,
     * and seek back to the beginning.  If any of these fails, die.
     */
    if (fseek(f, 0, SEEK_END))
        die("Unable to seek to end of file.");
    if (fgetpos(f, &len))
        die("Unable to get length of file.");

    if (fseek(f, 0, SEEK_SET))
        die("Unable to seek to beginning of file.");

    /* Allocate enough space or die trying */
    plain_text = (char *) malloc(len + 1);
    if (!plain_text)
        die("Unable to allocate memory for file plain_text.txt.");

    /* Read the data */
    if (!fread(plain_text, 1, len, f))
        die("Unable to read entire file plain_text.txt.");

    /* Close the file */
    fclose(f);

    /* Return the plaintext */
    return plain_text;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/****************************************
 * Get sequence for a given n, c
 ****************************************/
/*
 * This implements a portion of the Towers of Hanoi problem, finding
 * the state (recursively) for a given number of disks and number of steps.
 *
 * PSEUDOCODE:
 * PROCEDURE getSequence(configuration_num, num_disks, from_tower, to_tower,
spare_tower)
 *   IF (n = 0)
 *      do nothing
 *   IF (c < 2^(n-1))
 *      OUTPUT from_tower
 *      getSequence(configuration_num, n-1, from_tower, spare_tower, to_tower)
 *   ELSE
 *      OUTPUT to_tower
 *      getSequence(configuration_num - 2^(n-1), n-1, spare_tower, to_tower,
from_tower)
 */

int getSequence(long int c, long int n,
                const char *from_tower, const char *to_tower, const char
*spare_tower,
                char *result)
{
    if (!n)
    {
        /* Base case -- do nothing */
        return 1;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        }
        if (c < (long int) pow(2, n-1)) /* First half */
        {
            /* Big disk is on from_tower */
            strcat(result, from_tower);

            /* Move remaining disks from from_tower to spare_tower */
            getSequence(c, n-1, from_tower, spare_tower, to_tower, result);
        }
        else /* c >= pow(2, n-1) -- second half */
        {
            /* Big disk on to_tower */
            strcat(result, to_tower);

            /*  Move remaining disks from spare_tower to to_tower */
            getSequence(c - (long int) pow(2, n-1), n-1, spare_tower, to_tower,
from_tower, result);
        }
        return 1;
}
/****************************************
 * XOR on char *
 ****************************************/
/*
 * Assume a and b are strings containing only the characters
 * '1' and '0'.  The string result will be set to the
 * xor of a and b.  result will be as long as the minimum
 * length of a and b.  result is assumed to have enough space
 * allocated to hold this.
 */
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
int myXor(const char *a, const char *b, char *result)
{
    /* Count */
    int i;

    /* Our result will have a length equal to the min of lengths of a and b. */
    int len = ( strlen(a) > strlen(b) ? strlen(b) : strlen(a) );

    /* Walk the strings */
    for (i=0; i<len; ++i)
    {
        /* Compute XOR on characters '0' and '1' */
        if ((a[i] == '1' && b[i] == '0') ||
            (a[i] == '0' && b[i] == '1'))
            result[i] = '1';
        else
            result[i] = '0';
    }
    /* Null-terminate the result */
    result[len] = 0;\

    return 1;
}

/****************************************
 * Huffman Encode
 ****************************************/
/*
 * This is a special Huffman encoding. The alphabet is
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 * exactly three letters, with representations "0", "10", and "11".
 */

int myHuffman(const char *data,
                    const char *tower_0,
                    const char *tower_1,
                    const char *tower_2,
                    char *result)
{
     /* Get freqeuencies */
     int freq[3] = {0,0,0};

     /* The three outputs */
     char code[3][3];

     /* Get the "letter" length and data length */
     int letter_len = strlen(tower_0);
     int data_len = strlen(data);

     /* counter */
     int i;

     /* buffer */
     char *buf = (char *) malloc(letter_len + 2);

     /* Calculate frequencies */
     for (i=0; i<data_len; i += letter_len)
     {
          /* Copy the current letter to buf */
          strncpy(buf, data+i, letter_len);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        buf[letter_len] = 0;

        /* Which letter is it?  Increment that frequency*/
        if (!strcmp(buf, tower_0))
            ++freq[0];
        else if (!strcmp(buf, tower_1))
            ++freq[1];
        else if (!strcmp(buf, tower_2))
            ++freq[2];
        else
            die("Bad string passed to Huffman.");
    }
    /* Make the code -- hardcode the six possibilities */
    if (freq[0] >= freq[1] && freq[1] >= freq[2])
    { strcpy(code[0], "1"); strcpy(code[1], "01"); strcpy(code[2], "00"); }
    else if (freq[0] >= freq[2] && freq[2] >= freq[1])
    { strcpy(code[0], "1"); strcpy(code[2], "01"); strcpy(code[1], "00"); }
    else if (freq[1] >= freq[0] && freq[0] >= freq[2])
    { strcpy(code[1], "1"); strcpy(code[0], "01"); strcpy(code[2], "00"); }
    else if (freq[1] >= freq[2] && freq[2] >= freq[0])
    { strcpy(code[1], "1"); strcpy(code[2], "01"); strcpy(code[0], "00"); }
    else if (freq[2] >= freq[1] && freq[1] >= freq[0])
    { strcpy(code[2], "1"); strcpy(code[1], "01"); strcpy(code[0], "00"); }
    else if (freq[2] >= freq[0] && freq[0] >= freq[1])
    { strcpy(code[2], "1"); strcpy(code[0], "01"); strcpy(code[1], "00"); }

    /* Compute the result */
    for (i=0; i<data_len; i += letter_len)
    {
        /* Again, copy the letter to buf */
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            strncpy(buf, data+i, letter_len);
            buf[letter_len] = 0;

            /* This time, concatenate the appropriate code into result */
            if (!strcmp(buf, tower_0))
                  strcat(result, code[0]);
            else if (!strcmp(buf, tower_1))
                  strcat(result, code[1]);
            else if (!strcmp(buf, tower_2))
                  strcat(result, code[2]);
            else
                  die("Bad string passed to Huffman.");
      }
      /* Free the allocated buffer */
      free(buf);
      /* All done now */
      return 1;
}

/****************************************
 * Kasiski
 ****************************************/
/* Search a string for surprisingly close repetitions of substrings */

/*
 * Utility function.  Find the distance to substring chunk in data.
 * data_len=strlen(data) and chunk_len=strlen(chunk) are precomputed and
 * passed in, since this function is called many times.  Return -1 if
 * chunk does not appear in data.
*/
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
long int kasiskiSearch(char *data, int data_len, char *chunk, int chunk_len)
{
        /* Position count */
        int i;

        /* Walk the string, looking for chunk.  Must be byte-aligned. */
        for (i=0; i<data_len - chunk_len; i += 8)
                if (!strncmp(data+i, chunk, chunk_len))
                        return I;

        /* Failure */
        return -1;
}

/*
 * Our Kasiski test.  Search only byte-aligned.
 * -- data is the data to run Kasiski.
 * -- chunk_len is the length of substrings to search (should be multiple of 8)
 * -- alpha is fraction of expected distance for reporting.
 */

int kasiski(char *data, int chunk_len, double alpha)
{
        /* Count */
        int i;

        /* Allocate space for the chunk */
        char *chunk = (char *) malloc(chunk_len + 1);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/* Compute the data length*/
int data_len = strlen(data);

/* Compute the expected distance (2^chunk_len) */
long int expected_distance = (long int) pow(2, chunk_len);

/* Walk the data byte by byte */
for (i=0; i<data_len-chunk_len; i+=8)
{
    /* found distance */
    long int distance;

    /* Build the current chunk */
    strncpy(chunk, data+i, chunk_len);
    chunk[chunk_len] = 0;

    /* Find the distance */
    distance = kasiskiSearch( data + i + chunk_len , data_len - i - chunk_len,
                              chunk, chunk_len);

    /*
     * If we found the chunk, AND
     * it's surprisingly close, output a line of data
     */
    if (distance != -1 &&
        (double) distance / (double) expected_distance < alpha )
        printf("%d\t%d\t%ld\t%s\n", chunk_len/8, i/8, distance/8, chunk);
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        /* Free allocated chunk */
        free(chunk);

        /* All done.  Return success */
        return 1;
}

/***************************************
 * Chi-square
 ***************************************/
/* Run a chi-square test on 8, 16, and 24-bit chunks of data */

int chi(const char *data)
{
        /* Count of each character */
        double count[256];

        /* Counters */
        int i,j;

        /* Data length */
        int data_len = strlen(data);

        /* n = data_len/8 */
        double n = (double) data_len / (double) 8;

        /* Expected frequency of each letter */
        double expected = n / (double) 256;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/* Precompute some powers */
int pows[8] =
{ 1,        2,          4,        8,
  16,       32,          64,    128 };

/* The chi-square statistic */
double V = 0;

/* Initialize counts to zero */
for (i=0; i<256; ++i)
      count[i] = 0;

/* Walk the data by bytes */
for (i=0; i<data_len; i+=8)
{
      /* Figure out which byte to increment */
      int pos=0;
      for (j=0; j<8; ++j)
            if (data[i+j] == '1')
                  pos += pows[j];
      count[pos] += 1;
}
/* Output our counts */
printf("i\tcount\n");
for (i=0; i<256; ++i)
      printf("%d\t%f\n", i, count[i]);

/* Compute the chi-square statistic */
for (i=0; i<256; ++i)
      V += (count[i] - expected) * (count[i] - expected) / expected;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    /* Output the results */
    printf("n=%f V=%f\n", n, V);

    return 1;
}
/**************************************
 **************************************
 * Main Program
 **************************************
 **************************************/
int main(char **argv, char **envp)
{
    /* Representations for the three towers (private key) */
    const char *from_tower = "1010";
    const char *to_tower = "1100";
    const char *spare_tower = "0011";

    /* Primes p and q (private key) */
    long int p = 31121;
    long int q = 31183;

    /*
     * Starting value for x.  (private key)
     * We start x at (p+q) / 4, to ensure that x^2 will be greater than
     * either p or q (i.e. a_p and a_q will probably be different immediately.)
     */
    long int start_x = (p + q) / 4;

    /* Number of disks */
    long int n = 15;
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/* Plain text */
char *plain_ascii = getPlainAscii();

/*
 * Plain binary (only characters '1' and '0'). We precompute the length
 * for efficiency, but this is still a null-terminated string
 */

char *plain_bin;
long int plain_bin_len = 0;

/* Key buffer. Again we keep track of length for efficiency. */
char *Padluc_key;

/* Cyphertext */
char *cypher_bin;

/* Sequence buffers. */
char *buf_p;        /* Buffer for prime p */
char *buf_p_h;      /* Huffman encoded buffer for prime p */
char *buf_q;        /* Buffer for prime q */
char *buf_q_h;      /* Huffman encoded buffer for prime q */
char *buf;          /* General buffer (i.e. buf_p XOR buf_q */
char *buf_h;        /* Huffman encoded buf */

/*
 * Iterate using x. We take a_p and a_q to be quadratic residues of x mod p and q,
 * respectively.
 */
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
long int x = start_x;
long int a_p = 1;
long int a_q = 1;

/* Timing variables */
clock_t start_time = 0;
clock_t finish_time = 0;

/* Counters */
int j;

/* Translate the plainAscii into plainBin, the binary represenation */
plain_bin = (char *) malloc(8 * (strlen(plain_ascii) + 1));
asciiToBin(plain_ascii, plain_bin);
plain_bin_len = strlen(plain_bin);

/*
 * Allocate enough space for the key.  The key may be as much as
 * 2*n*tower_rep_len longer than the plaintext.  Add a little extra for safety.
 * Initialize to the empty string.  Allocate similar space for the cyphertext
 */

Padluc_key = (char *) malloc(strlen(plain_bin) + 2 * n * strlen(from_tower));
Padluc_key[0] = 0;
cypher_bin = (char *) malloc(strlen(plain_bin) + 2 * n * strlen(from_tower));
cypher_bin[0] = 0;

/*
 * Allocate enough space for the sequence buffers.  They require 2*n*tower_rep_len
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
 * characters, plus one for null-termination
 */

buf_p   = (char *) malloc(n * strlen(from_tower) + 1);
buf_p_h = (char *) malloc(n * strlen(from_tower) + 1);
buf_q   = (char *) malloc(n * strlen(from_tower) + 1);
buf_q_h = (char *) malloc(n * strlen(from_tower) + 1);
buf     = (char *) malloc(n * strlen(from_tower) + 1);
buf_h   = (char *) malloc(n * strlen(from_tower) + 1);

/* Start timing now */
start_time = clock();

/*
 * Main algorithm:  while we need more key... (note that the various
 * keys might be of different lengths, since Huffman changes the length)
 */

while ( (long int) strlen(Padluc_key) < plain_bin_len )
{
      /* Initialize buffers */
      buf_p[0] = 0;
      buf_p_h[0] = 0;
      buf_q[0] = 0;
      buf_q_h[0] = 0;
      buf[0] = 0;

      x++;

      /* Make sure that x hasn't cycled */
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        if (x > (long int) pow(2, n) - 1)
            die("n too small for current plaintext.");

        /* Compute quadratic residues for p and q */
        a_p = (long int) x*x % (long int) p;
        a_q = (long int) x*x % (long int) q;
        if (a_p < 0 || a_q < 0)
            die("Overflow.");

        /* Compute the sequences a_p and a_q */
        getSequence(a_p, n, from_tower, to_tower, spare_tower, buf_p);
        getSequence(a_q, n, from_tower, to_tower, spare_tower, buf_q);

        /* Huffman encode the sequences */
        myHuffman(buf_p, from_tower, to_tower, spare_tower, buf_p_h);
        myHuffman(buf_q, from_tower, to_tower, spare_tower, buf_q_h);

        /* XOR the sequences */
        myXor(buf_p, buf_q, buf);
        myXor(buf_p_h, buf_q_h, buf_h);

        /* Concatenate the result. */
        strcat(Padluc_key, buf_h);
    }

    /* XOR with the plaintext to get the cyphertext */
    myXor(Padluc_key, plain_bin, cypher_bin);

    /* Finish timing now and output */
    finish_time = clock();
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
/* printf("Time: %0.4f\n",
                ( (double) finish_time - (double) start_time) / (double)
CLOCKS_PER_SEC); */
    /* Output the various results */
    printf("Plaintext:\n%s\n\nKey:\n%s\n\nCyphertext:\n%s\n\n",
            plain_bin, Padluc_key, cypher_bin);

    /* Output a chi-square test on one-byte chunks for each. */
    printf("Chi-square test for cypher_bin\n");
    chi(cypher_bin);

    /*
     * Kasiski search each cypher_bin for suprising results.
     * Define "surprising" as 1/100 as close together as expected.
     * Only go up to 128-bit chunks -- if we find any of those, we're
     * pretty sure something's wrong.
     */

    printf("Kasiski for cypher_bin\n");
    printf("len\tpos\tdist\tchunk\n");
    for (j=16; j<128; j+=8)
        kasiski(cypher_bin, j, 0.01);

    /* Be good!  Free all dynamically allocated space.*/
    free(Padluc_key);
    free(cypher_bin);
    free(plain_ascii);
    free(plain_bin);
    free(buf_p);
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    free(buf_p_h);
    free(buf_q);
    free(buf_q_h);
    free(buf);
    free(buf_h);
    /* Completed */
    return 0;
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A.8 Chi-square and Kasiski Test Results for Four Alternative Padluc Algorithms

Plaintext:

Some English text, followed by a similar length of repetition of the character "X". (The second part tests encryption of a long sequence of the same character.) Plaintext length: 6601 characters.

Chi-Square Test:

Chi-square run by comparing counts of each byte of ciphertext with the "expected" count, 6601/256 ~ 25.8. Chi square results:

Test results:

Algorithm 1: V = 32,373

Algorithm 2: V = 362

Algorithm 3: V = 2,688

Algorithm 4: V = 57,390

Analysis of the test results:

In all cases, chi-square is above the 99 percentile, although algorithm 2 and to a lesser extent algorithm 3 have "reasonable" counts. For 255 degrees of freedom (number of possibly bytes - 1), the 99% probability for chi-square comes at roughly 332. Recall that algorithms 2 and 3 use Huffman encoding for 2 and 1 primes, respectively.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Kasiski Test:

Kasiski performed by searching for repeated polygrams in 2, 3, ...,16-character substrings in the ciphertext, and outputting a result when these repetitions were "surprisingly" close together, where for this test "surprising" was defined as "less than 1% of the expected distance."

Algorithm 1:

For the first chunk of the ciphertext, there are indications of a possible cycle with a factor of 7, but there are not really too many repetitions -- only ~20 of length 2 and 3, with a probability of 1% doesn't seem surprising. However, the second chunk of ciphertext (encrypting repetitions of "X") shows pathological behavior, with a tremendous number of very close repetitions of substrings of length 2 and 3 (thousands of them). This fits well with the chi-square, which says that the counts for these bytes are all wrong.

Algorithm 2:

Only 10 "surprising" repetitions, all except one of length 2 (that one of length 3) and with no particular pattern in the distances. This is the only algorithm that passes the Kasiski test.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Algorithm 3:

59 surprising repetitions of length 2, 21 of length 3, which seems to indicate possible patterns. Again, the frequency goes dramatically up when encrypting pathological plaintext (the second half).

Algorithm 4:

Thousands of surprising repetitions (more than for algorithm1) with repetitions of four-character strings appearing. Again, this fits in with a very unequal distribution of patterns.

Overall, only algorithm 2 really appears to have passed both tests. The fact that Huffman changes the length of the key is very important, particularly in shifting alignment when two such codes are XORed, or when the Padluc key is XORed with plaintext.

We chose four 4-bit tower representations (1100, 0011, and 1010), the only possible substrings of the Padluc key for algorithm 4. XORing them in any combination gives 0000, 0110, and 1001 as the only possible substrings of the Padluc key for algorithm 1. Clearly with a long string of pathological plaintext, this will fail lots of statistical tests -- only 9 possible bytes can be generated, even if the sequences are completely random.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

BIBLIOGRAPHY

1. Afriat, S. N., "The Ring of Linked Rings", Duckworth, 1982.

2. Allardice, R. E. and Fraser, A. Y., "La Tour d'Hanoi", Proc. Edinburgh Math. Soc., v. 2, 1884, pp 50-53.

3. Allouche, J. P., Astoorian, D., Randall, J., Shallit, J., "Morphisms, Squarefree Strings and the Tower of Hanoi Puzzle", Amer. Math. Monthly, v. 101, 1994, pp 651-658.

4. Allouche, J. P., Dress, F., "Tours de Hanoi et Automates", Informatique Theorique et Applications, v. 24, no. 1, 1990, pp 1-15.

5. Atkinson, M. D., "The Cyclic Towers of Hanoi", Inform. Process. Lett., v.13, 1981, pp 118-119.

6. Berman, G., and Fryer, K.D., "Introduction to Combinatorics", Academic Press, 1972.

7. Cain, T. R., Sherman, A. T., "How to Break Gifford's Cipher", Cryptologia, v. 21, no. 3, 1997, pp 237- 286.

8. Charniak, E., and McDermott, D., "Introduction to Artificial Intelligence", Addison-Wesley, 1986.

9. Crowe, D. W., "The n-Dimensional Cube and the Towers of Hanoi", Amer. Math. Monthly, v. 63, 1956, pp 29-30.

10. Cull, P. and Ecklund Jr., E. F., "Towers of Hanoi and Analysis of Algorithm", Amer. Math. Monthly, v. 92, 1985, pp 407-420.

11. Dijkstra, E. W., "A Short Introduction to the Art of Programming", EWD 316, 1971.

12. Dudeney, H. D., "The Canterbury Puzzles and Other Curious Problems", E. P. Dutton, 1908; 4th Edition, Dover Publications, Inc., 1958.

13. Er, M. C., "The Tower of Hanoi Problem -- A Reply", Comput. J., v. 27, 1984, pp 285.

14. ------, "A Representation Approach to the Towers of Hanoi Problem", Comput. J., v. 25, 1982, pp 442-447.

15. ------, "Performance Evaluation of Recursive and Iterative Algorithms For the Towers of Hanoi Problem", Computing, v. 37, (1986), 93-102.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

16. ------, "The Cyclic Towers of Hanoi: A Representation Approach", Comput. J., v. 27, 1984, pp 171-175.

17. ------, "A Generalization of the Cyclic Towers of Hanoi: An Iterative Solution", Internat. J. Comput. Math., v. 15, 1984, pp 129-140.

18. ------, "The Complexity of the Generalized Cyclic Towers of Hanoi Problem", J. Algorithms, v. 6, 1985, pp 351-358.

19. ------, "Mathematical Games: The Curious Properties of the Gray Code and How it Can Be Used to Solve Puzzles", Sci. Amer., August 1972, pp 106-109.

20. Gilman, D.W., Mohtashemi, M., Rivest, R.L., "On Breaking a Huffman Code", IEEE Trans. Inform. Theory, v. 42, no. 3, 1996, pp 972-976.

21. Golomb, S. W., "Shift Register Sequences", Holden-Day Inc., 1967.

22. Graham, R. L., Knuth, D. E. and Patashink, O., "Concrete Mathematics", Addison-Wesley, 1989.

23. Hardy, W. H., and Wright, E. M., "An Introduction to the Theory of Numbers", Clarendon Press, 1954.

24. Harkin, D.,"On the Mathematical Work of Francois-Edouard-Anatole Lucas", Enseign. Math., v. 2, no. 3, 1957, pp 276-288.

25. Hayes, P. J., "Note on the Towers of Hanoi Problem", Comput. J., v. 20, 1977, pp 282-285.

26. Hinz, A. M., "The Tower of Hanoi", Enseign. Math., v. 35, 1989, pp 289-321.

27. ------, "Shortest Path Between Regular States of the Tower of Hanoi", Inform. Sci., (to appear).

28. ------, "Pascal's Triangle and the Tower of Hanoi", Amer. Math. Monthly, v. 99, 1992, pp 538-544.

29. Kahn, D., "The Codebreakers", Macmillan, 1967.

30. Klein, C. S., and Minsker, S., "The Super Towers of Hanoi Problem: Large Rings on Small Rings", Discrete Math., v. 114, 1993, pp 283-295.

31. Knuth, D. E., "The Art of Computer Programming: v. 2, Seminumerical Algorithms", 2nd Edition, Addison-Wesley, 1981.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

32. Kohavi, Z., "Switching and Finite Automata Theory", McGraw-Hill, 1978.

33. Kruse, R. L., "Data Structures and Program Design", Prentice-Hall, 1984.

34. Lu, X. M., "Towers of Hanoi Graphs", Internat. J. Comput. Math., v. 19, 1986, pp 23-38.

35. Lucas, E. A., "Theorie des Nombers", Gauthier-Villars, 1891.

36. Mandelbrot, B., "The Fractal Geometry of Nature", Freeman, 1977.

37. Maurer, S. B., and Ralston, A., "Discrete Algorithmic Mathematics", Addison-Wesley, 1990.

38. Minsker, S., "The Towers of Hanoi Rainbow Problem: Coloring the Rings", J. Algorithms, v. 10, 1989, pp 1-19.

39. Narkiewicz, W., "Classical Problems in Number Theory", PWN Polish Scientific Publishers, 1986.

40. Ore, O., "Invitation to Number Theory", Random House, 1967.

41. Parville, H. de, "La Tour d'Hanoi et la Question du Tonkin", La Nature, v. 12, 1884, pp 285-286.

42. Paull, M. C., "Algorithm Design, A Recursion Transformation Framework", John Wiley & Sons, 1988.

43. Poole, D. G., "The Towers and Triangles of Professor Claus (or, Pascal Knows Hanoi)", Math. Magazine, v. 67, 1994, pp 324-344.

44. Rohl, J. S., "Recursion via Pascal", Cambridge University Press, 1984.

45. Scorer, R. S., Grundy, P. M. and Smith, C. A. B., "Some Binary Games", Math. Gaz., v. 28, 1944, pp 96-103.

46. Schneier, B., "Applied Cryptography", John Wiley & Sons, 1996.

47. Schroeder, M., "Number Theory in Science and Communication", Springer-Verlag, 1984.

48. Sierpiński, W., "Elementary Theory of Numbers", Panstwowe Wydawnictwo Naukowe, 1964.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

49. Simon, H. A., "The Functional Equivalence of Problem Solving Skills", Cognitive Psychology, v. 7, 1975, pp 268-288.

50. Spencer, D., "Exploring Number Theory with Microcomputers", Camelot Publishing Company, 1989.

51. Stallings, W., "Network and Internetwork Security", Prentice Hall, 1995.

52. Stewart, I., "Four Encounters with Sierpinski's Gasket", Math. Intelligencer, v. 17, 1995, pp 52-64.

53. ------, "Les Fractals de Pascal", Pour la Science, v. 129, 1988, pp 100-104.

54. ------, "Le Lion, la Lama et la Laitue", Pour la Science, v. 142, 1989, pp 102-107.

55. Stinson, D. R., "Cryptography: Theory and Practice", CRC Press, 1995.

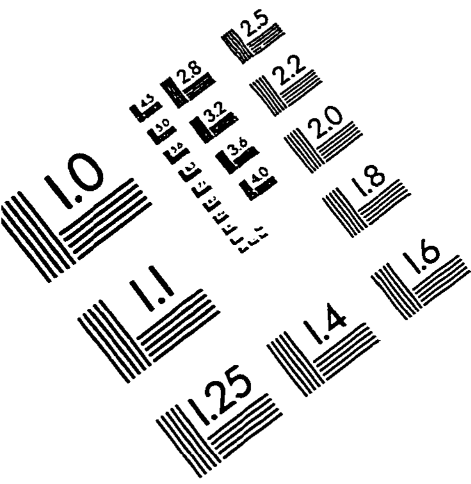
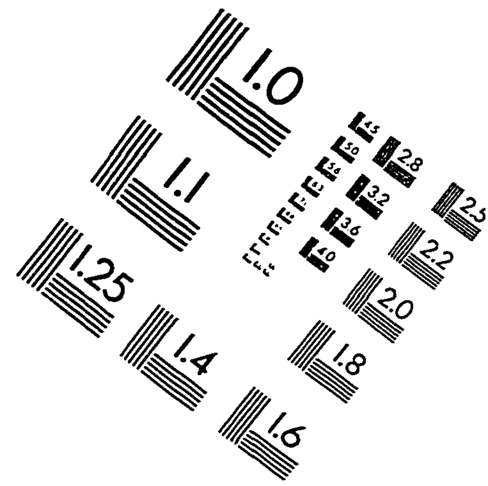
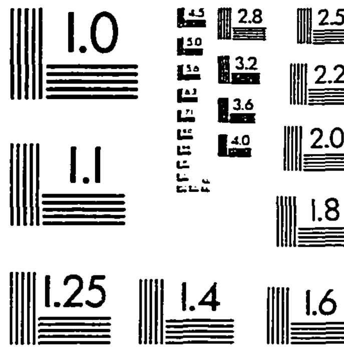
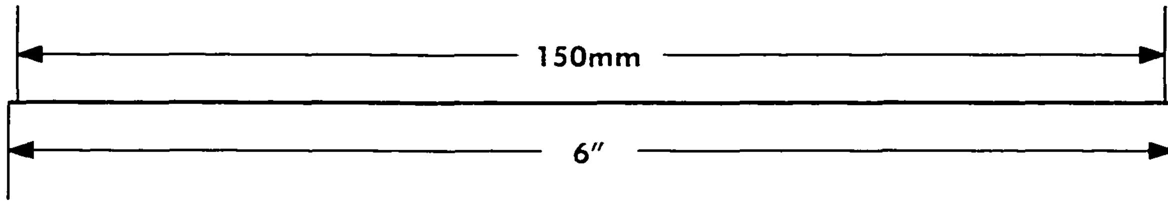
56. Walsh, T. R., "The Towers of Hanoi Revisited -- Moving the Rings By Counting the Moves", Inform. Process Lett, v. 15, 1982, pp 64-67.

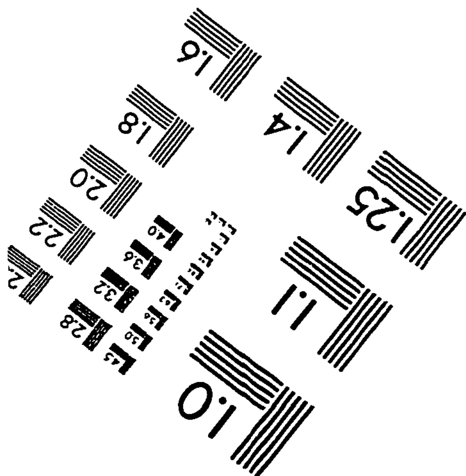
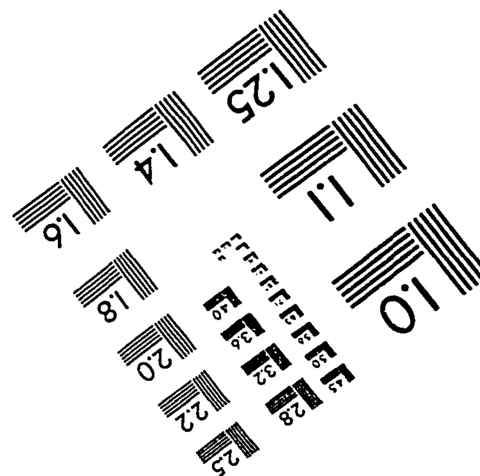
57. Walsh, T. R., " The Generalized Towers of Hanoi for Space-Deficient Computers and Forgetful Humans", Math. Intelligencer, v. 20, 1998, pp 32-38.

58. Wood, D., "The Towers of Brahma and Hanoi Revisited", J. Recreational Math., v. 14, 1981, pp 17-24.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE . Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.