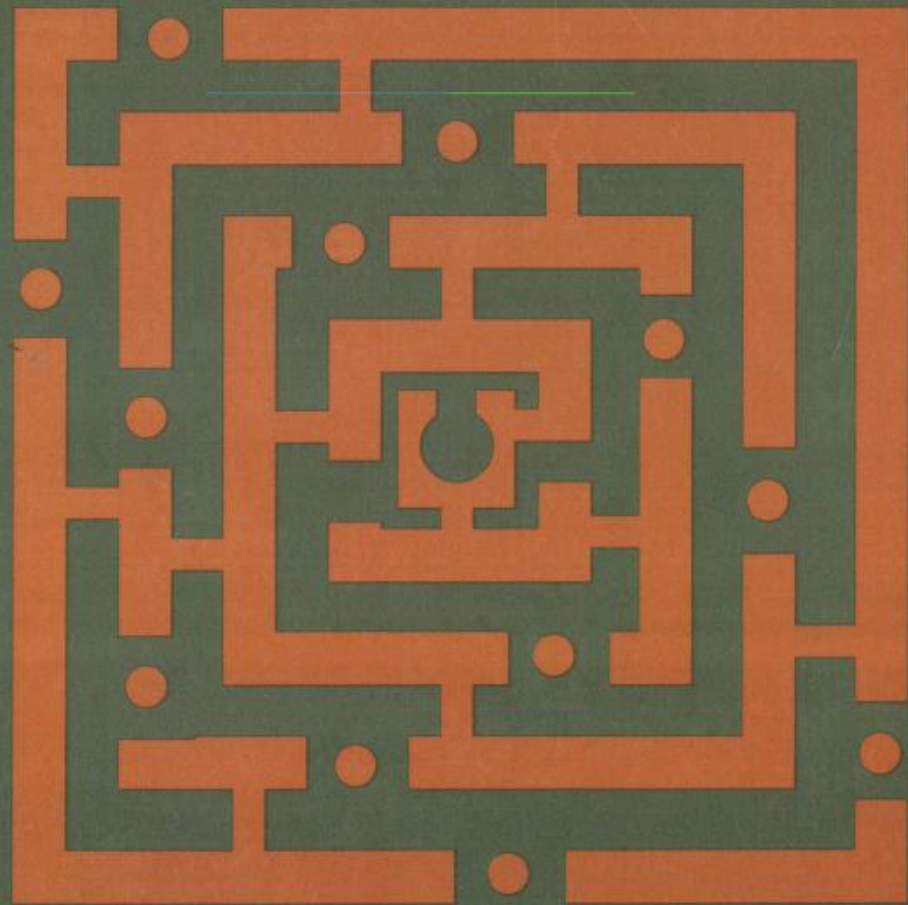


HANDBOOK of APPLIED CRYPTOGRAPHY



Alfred J. Menezes
Paul C. van Oorschot
Scott A. Vanstone

Facebook's Exhibit No. 1005
0001

MOBILEIRON, INC. - EXHIBIT 1011
Page 001

Computer Sciences
Applied Mathematics
Engineering

HANDBOOK of APPLIED CRYPTOGRAPHY

LIBRARY OF CONGRESS



0 013 663 411 2

Cryptography, in particular public-key cryptography, has emerged in the last 20 years as an important discipline that is not only the subject of an enormous amount of research, but provides the foundation for information security in many applications. Standards are emerging to meet the demands for cryptographic protection in most areas of data communications. Public-key cryptographic techniques are now in widespread use in industry, especially in the financial services industry, in the public sector, and by individuals for their personal privacy, such as in electronic mail. This Handbook will serve as a valuable reference for the novice as well as for the expert who needs a wider scope of coverage within the area of cryptography. It is a necessary and timely guide for professionals who practice the art of cryptography.

The *Handbook of Applied Cryptography* provides a treatment that is multifunctional:

- It serves as an introduction to the more practical aspects of both conventional and public-key cryptography
- It is a valuable source of the latest techniques and algorithms for the serious practitioner
- It provides an integrated treatment of the field, while still presenting each major topic as a self-contained unit
- It provides a mathematical treatment to accompany practical discussions
- It contains enough abstraction to be a valuable reference for theoreticians while containing enough detail to actually allow implementation of the algorithms discussed

This is the definitive cryptography reference that novice as well as experienced developers, designers, researchers, engineers, computer scientists, and mathematicians alike will find indispensable.

8523

ISBN 0-8493-8523-7



9 780849 385230

Facebook's Exhibit No. 1005
0002

MOBILEIRON, INC. - EXHIBIT 1011

Page 002

HANDBOOK of APPLIED CRYPTOGRAPHY

Menezes

FT MEADE
GenColl

QA 76

.9


.A25

M463

1997

Facebook's Exhibit No. 1005
0003

MOBILEIRON, INC. - EXHIBIT 1011
Page 003



HANDBOOK of APPLIED CRYPTOGRAPHY

Facebook's Exhibit No. 1005
0004

The CRC Press Series on
**DISCRETE
MATHEMATICS
AND
ITS APPLICATIONS**

Series Editor
Kenneth H. Rosen, Ph.D.
AT&T Bell Laboratories

*Charles J. Colbourn and Jeffery H. Dinitz, The CRC
Handbook of Combinatorial Designs*

*Steven Furino, Ying Miao, and Jianxing Yin,
Frames and Resolvable Designs:
Uses, Constructions, and Existence*

*Daryl D. Harms, Miroslav Kraetzl, Charles J. Colbourn,
and Stanley J. Devitt, Network Reliability:
Experiments with A Symbolic Algebra Environment*

Richard A. Mollin, Quadratics

Douglas R. Stinson, Cryptography: Theory and Practice

HANDBOOK of APPLIED CRYPTOGRAPHY

Alfred J. Menezes
Paul C. van Oorschot
Scott A. Vanstone



CRC Press

Boca Raton New York London Tokyo

Facebook's Exhibit No. 1005
0006

MOBILEIRON, INC. - EXHIBIT 1011
Page 006

QA76
.9
A25 M463
1997
SCIR



Library of Congress Cataloging-in-Publication Data

Menezes, A. J. (Alfred J.), 1965-
Handbook of applied cryptography / Alfred Menezes, Paul van Oorschot,
Scott Vanstone.
p. cm. -- (CRC Press series on discrete mathematics and its
applications)
Includes bibliographical references and index.
ISBN 0-8493-8523-7 (alk. paper)
1. Computers--Access control--Handbooks, manuals, etc.
2. Cryptography--Handbooks, manuals, etc. I. Van Oorschot, Paul C.
II. Vanstone, Scott A. III. Title. IV. Series: Discrete
mathematics and its applications.
QA76.9.A25M463 1996
005.8'2--dc20

96-27609
CIP

This book contains information obtained from authentic and highly regarded sources. Reprinted material is quoted with permission, and sources are indicated. A wide variety of references are listed. Reasonable efforts have been made to publish reliable data and information, but the author and the publisher cannot assume responsibility for the validity of all materials or for the consequences of their use.

Neither this book nor any part may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, microfilming, and recording, or by any information storage or retrieval system, without prior permission in writing from the publisher.

The consent of CRC Press does not extend to copying for general distribution, for promotion, for creating new works, or for resale. Specific permission must be obtained in writing from CRC Press for such copying.

Direct all inquiries to CRC Press, Inc., 2000 Corporate Blvd., N.W., Boca Raton, Florida 33431.

© 1997 by CRC Press, Inc.

No claim to original U.S. Government works
International Standard Book Number 0-8493-8523-7
Library of Congress Card Number 96-27609
Printed in the United States of America 1 2 3 4 5 6 7 8 9 0
Printed on acid-free paper

To Archie and Lida Menezes

To Cornelis Henricus van Oorschot
and Maria Anna Buys van Vugt

To Margret and Gordon Vanstone

Contents in Brief

Table of Contents	v
List of Tables	xv
List of Figures	xix
Foreword	xxi
Preface	xxiii
1 Overview of Cryptography	1
2 Mathematical Background	49
3 Number-Theoretic Reference Problems	87
4 Public-Key Parameters	133
5 Pseudorandom Bits and Sequences	169
6 Stream Ciphers	191
7 Block Ciphers	223
8 Public-Key Encryption	283
9 Hash Functions and Data Integrity	321
10 Identification and Entity Authentication	385
11 Digital Signatures	425
12 Key Establishment Protocols	489
13 Key Management Techniques	543
14 Efficient Implementation	591
15 Patents and Standards	635
A Bibliography of Papers from Selected Cryptographic Forums	663
References	703
Index	755

Table of Contents

List of Tables	xv
List of Figures	xix
Foreword by R.L. Rivest	xxi
Preface	xxiii
1 Overview of Cryptography	1
1.1 Introduction	1
1.2 Information security and cryptography	2
1.3 Background on functions	6
1.3.1 Functions (1-1, one-way, trapdoor one-way)	6
1.3.2 Permutations	10
1.3.3 Involutions	10
1.4 Basic terminology and concepts	11
1.5 Symmetric-key encryption	15
1.5.1 Overview of block ciphers and stream ciphers	15
1.5.2 Substitution ciphers and transposition ciphers	17
1.5.3 Composition of ciphers	19
1.5.4 Stream ciphers	20
1.5.5 The key space	21
1.6 Digital signatures	22
1.7 Authentication and identification	24
1.7.1 Identification	24
1.7.2 Data origin authentication	25
1.8 Public-key cryptography	25
1.8.1 Public-key encryption	25
1.8.2 The necessity of authentication in public-key systems	27
1.8.3 Digital signatures from reversible public-key encryption	28
1.8.4 Symmetric-key vs. public-key cryptography	31
1.9 Hash functions	33
1.10 Protocols and mechanisms	33
1.11 Key establishment, management, and certification	35
1.11.1 Key management through symmetric-key techniques	36
1.11.2 Key management through public-key techniques	37
1.11.3 Trusted third parties and public-key certificates	39
1.12 Pseudorandom numbers and sequences	39
1.13 Classes of attacks and security models	41
1.13.1 Attacks on encryption schemes	41
1.13.2 Attacks on protocols	42
1.13.3 Models for evaluating security	42
1.13.4 Perspective for computational security	44
1.14 Notes and further references	45

2	Mathematical Background	49
2.1	Probability theory	50
2.1.1	Basic definitions	50
2.1.2	Conditional probability	51
2.1.3	Random variables	51
2.1.4	Binomial distribution	52
2.1.5	Birthday attacks	53
2.1.6	Random mappings	54
2.2	Information theory	56
2.2.1	Entropy	56
2.2.2	Mutual information	57
2.3	Complexity theory	57
2.3.1	Basic definitions	57
2.3.2	Asymptotic notation	58
2.3.3	Complexity classes	59
2.3.4	Randomized algorithms	62
2.4	Number theory	63
2.4.1	The integers	63
2.4.2	Algorithms in \mathbb{Z}	66
2.4.3	The integers modulo n	67
2.4.4	Algorithms in \mathbb{Z}_n	71
2.4.5	The Legendre and Jacobi symbols	72
2.4.6	Blum integers	74
2.5	Abstract algebra	75
2.5.1	Groups	75
2.5.2	Rings	76
2.5.3	Fields	77
2.5.4	Polynomial rings	78
2.5.5	Vector spaces	79
2.6	Finite fields	80
2.6.1	Basic properties	80
2.6.2	The Euclidean algorithm for polynomials	81
2.6.3	Arithmetic of polynomials	83
2.7	Notes and further references	85
3	Number-Theoretic Reference Problems	87
3.1	Introduction and overview	87
3.2	The integer factorization problem	89
3.2.1	Trial division	90
3.2.2	Pollard's rho factoring algorithm	91
3.2.3	Pollard's $p - 1$ factoring algorithm	92
3.2.4	Elliptic curve factoring	94
3.2.5	Random square factoring methods	94
3.2.6	Quadratic sieve factoring	95
3.2.7	Number field sieve factoring	98
3.3	The RSA problem	98
3.4	The quadratic residuosity problem	99
3.5	Computing square roots in \mathbb{Z}_n	99
3.5.1	Case (i): n prime	100
3.5.2	Case (ii): n composite	101

3.6	The discrete logarithm problem	103
3.6.1	Exhaustive search	104
3.6.2	Baby-step giant-step algorithm	104
3.6.3	Pollard's rho algorithm for logarithms	106
3.6.4	Pohlig-Hellman algorithm	107
3.6.5	Index-calculus algorithm	109
3.6.6	Discrete logarithm problem in subgroups of \mathbb{Z}_p^*	113
3.7	The Diffie-Hellman problem	113
3.8	Composite moduli	114
3.9	Computing individual bits	114
3.9.1	The discrete logarithm problem in \mathbb{Z}_p^* — individual bits	116
3.9.2	The RSA problem — individual bits	116
3.9.3	The Rabin problem — individual bits	117
3.10	The subset sum problem	117
3.10.1	The L^3 -lattice basis reduction algorithm	118
3.10.2	Solving subset sum problems of low density	120
3.10.3	Simultaneous diophantine approximation	121
3.11	Factoring polynomials over finite fields	122
3.11.1	Square-free factorization	123
3.11.2	Berlekamp's Q -matrix algorithm	124
3.12	Notes and further references	125
4	Public-Key Parameters	133
4.1	Introduction	133
4.1.1	Generating large prime numbers naively	134
4.1.2	Distribution of prime numbers	134
4.2	Probabilistic primality tests	135
4.2.1	Fermat's test	136
4.2.2	Solovay-Strassen test	137
4.2.3	Miller-Rabin test	138
4.2.4	Comparison: Fermat, Solovay-Strassen, and Miller-Rabin	140
4.3	(True) Primality tests	142
4.3.1	Testing Mersenne numbers	142
4.3.2	Primality testing using the factorization of $n - 1$	143
4.3.3	Jacobi sum test	144
4.3.4	Tests using elliptic curves	145
4.4	Prime number generation	145
4.4.1	Random search for probable primes	145
4.4.2	Strong primes	149
4.4.3	NIST method for generating DSA primes	150
4.4.4	Constructive techniques for provable primes	152
4.5	Irreducible polynomials over \mathbb{Z}_p	154
4.5.1	Irreducible polynomials	154
4.5.2	Irreducible trinomials	157
4.5.3	Primitive polynomials	157
4.6	Generators and elements of high order	160
4.6.1	Selecting a prime p and generator of \mathbb{Z}_p^*	164
4.7	Notes and further references	165

5	Pseudorandom Bits and Sequences	169
5.1	Introduction	169
5.1.1	Background and Classification	170
5.2	Random bit generation	171
5.3	Pseudorandom bit generation	173
5.3.1	ANSI X9.17 generator	173
5.3.2	FIPS 186 generator	174
5.4	Statistical tests	175
5.4.1	The normal and chi-square distributions	176
5.4.2	Hypothesis testing	179
5.4.3	Golomb's randomness postulates	180
5.4.4	Five basic tests	181
5.4.5	Maurer's universal statistical test	183
5.5	Cryptographically secure pseudorandom bit generation	185
5.5.1	RSA pseudorandom bit generator	185
5.5.2	Blum-Blum-Shub pseudorandom bit generator	186
5.6	Notes and further references	187
6	Stream Ciphers	191
6.1	Introduction	191
6.1.1	Classification	192
6.2	Feedback shift registers	195
6.2.1	Linear feedback shift registers	195
6.2.2	Linear complexity	198
6.2.3	Berlekamp-Massey algorithm	200
6.2.4	Nonlinear feedback shift registers	202
6.3	Stream ciphers based on LFSRs	203
6.3.1	Nonlinear combination generators	205
6.3.2	Nonlinear filter generators	208
6.3.3	Clock-controlled generators	209
6.4	Other stream ciphers	212
6.4.1	SEAL	213
6.5	Notes and further references	216
7	Block Ciphers	223
7.1	Introduction and overview	223
7.2	Background and general concepts	224
7.2.1	Introduction to block ciphers	224
7.2.2	Modes of operation	228
7.2.3	Exhaustive key search and multiple encryption	233
7.3	Classical ciphers and historical development	237
7.3.1	Transposition ciphers (background)	238
7.3.2	Substitution ciphers (background)	238
7.3.3	Polyalphabetic substitutions and Vigenère ciphers (historical)	241
7.3.4	Polyalphabetic cipher machines and rotors (historical)	242
7.3.5	Cryptanalysis of classical ciphers (historical)	245
7.4	DES	250
7.4.1	Product ciphers and Feistel ciphers	250
7.4.2	DES algorithm	252
7.4.3	DES properties and strength	256

7.5	FEAL	259
7.6	IDEA	263
7.7	SAFER, RC5, and other block ciphers	266
7.7.1	SAFER	266
7.7.2	RC5	269
7.7.3	Other block ciphers	270
7.8	Notes and further references	271
8	Public-Key Encryption	283
8.1	Introduction	283
8.1.1	Basic principles	284
8.2	RSA public-key encryption	285
8.2.1	Description	286
8.2.2	Security of RSA	287
8.2.3	RSA encryption in practice	290
8.3	Rabin public-key encryption	292
8.4	ElGamal public-key encryption	294
8.4.1	Basic ElGamal encryption	294
8.4.2	Generalized ElGamal encryption	297
8.5	McEliece public-key encryption	298
8.6	Knapsack public-key encryption	300
8.6.1	Merkle-Hellman knapsack encryption	300
8.6.2	Chor-Rivest knapsack encryption	302
8.7	Probabilistic public-key encryption	306
8.7.1	Goldwasser-Micali probabilistic encryption	307
8.7.2	Blum-Goldwasser probabilistic encryption	308
8.7.3	Plaintext-aware encryption	311
8.8	Notes and further references	312
9	Hash Functions and Data Integrity	321
9.1	Introduction	321
9.2	Classification and framework	322
9.2.1	General classification	322
9.2.2	Basic properties and definitions	323
9.2.3	Hash properties required for specific applications	327
9.2.4	One-way functions and compression functions	327
9.2.5	Relationships between properties	329
9.2.6	Other hash function properties and applications	330
9.3	Basic constructions and general results	332
9.3.1	General model for iterated hash functions	332
9.3.2	General constructions and extensions	333
9.3.3	Formatting and initialization details	334
9.3.4	Security objectives and basic attacks	335
9.3.5	Bitsizes required for practical security	337
9.4	Unkeyed hash functions (MDCs)	338
9.4.1	Hash functions based on block ciphers	338
9.4.2	Customized hash functions based on MD4	343
9.4.3	Hash functions based on modular arithmetic	351
9.5	Keyed hash functions (MACs)	352
9.5.1	MACs based on block ciphers	353

9.5.2	Constructing MACs from MDCs	354
9.5.3	Customized MACs	356
9.5.4	MACs for stream ciphers	358
9.6	Data integrity and message authentication	359
9.6.1	Background and definitions	359
9.6.2	Non-malicious vs. malicious threats to data integrity	362
9.6.3	Data integrity using a MAC alone	364
9.6.4	Data integrity using an MDC and an authentic channel	364
9.6.5	Data integrity combined with encryption	364
9.7	Advanced attacks on hash functions	368
9.7.1	Birthday attacks	369
9.7.2	Pseudo-collisions and compression function attacks	371
9.7.3	Chaining attacks	373
9.7.4	Attacks based on properties of underlying cipher	375
9.8	Notes and further references	376
10	Identification and Entity Authentication	385
10.1	Introduction	385
10.1.1	Identification objectives and applications	386
10.1.2	Properties of identification protocols	387
10.2	Passwords (weak authentication)	388
10.2.1	Fixed password schemes: techniques	389
10.2.2	Fixed password schemes: attacks	391
10.2.3	Case study – UNIX passwords	393
10.2.4	PINs and passkeys	394
10.2.5	One-time passwords (towards strong authentication)	395
10.3	Challenge-response identification (strong authentication)	397
10.3.1	Background on time-variant parameters	397
10.3.2	Challenge-response by symmetric-key techniques	400
10.3.3	Challenge-response by public-key techniques	403
10.4	Customized and zero-knowledge identification protocols	405
10.4.1	Overview of zero-knowledge concepts	405
10.4.2	Feige-Fiat-Shamir identification protocol	410
10.4.3	GQ identification protocol	412
10.4.4	Schnorr identification protocol	414
10.4.5	Comparison: Fiat-Shamir, GQ, and Schnorr	416
10.5	Attacks on identification protocols	417
10.6	Notes and further references	420
11	Digital Signatures	425
11.1	Introduction	425
11.2	A framework for digital signature mechanisms	426
11.2.1	Basic definitions	426
11.2.2	Digital signature schemes with appendix	428
11.2.3	Digital signature schemes with message recovery	430
11.2.4	Types of attacks on signature schemes	432
11.3	RSA and related signature schemes	433
11.3.1	The RSA signature scheme	433
11.3.2	Possible attacks on RSA signatures	434
11.3.3	RSA signatures in practice	435

11.3.4	The Rabin public-key signature scheme	438
11.3.5	ISO/IEC 9796 formatting	442
11.3.6	PKCS #1 formatting	445
11.4	Fiat-Shamir signature schemes	447
11.4.1	Feige-Fiat-Shamir signature scheme	447
11.4.2	GQ signature scheme	450
11.5	The DSA and related signature schemes	451
11.5.1	The Digital Signature Algorithm (DSA)	452
11.5.2	The ElGamal signature scheme	454
11.5.3	The Schnorr signature scheme	459
11.5.4	The ElGamal signature scheme with message recovery	460
11.6	One-time digital signatures	462
11.6.1	The Rabin one-time signature scheme	462
11.6.2	The Merkle one-time signature scheme	464
11.6.3	Authentication trees and one-time signatures	466
11.6.4	The GMR one-time signature scheme	468
11.7	Other signature schemes	471
11.7.1	Arbitrated digital signatures	472
11.7.2	ESIGN	473
11.8	Signatures with additional functionality	474
11.8.1	Blind signature schemes	475
11.8.2	Undeniable signature schemes	476
11.8.3	Fail-stop signature schemes	478
11.9	Notes and further references	481
12	Key Establishment Protocols	489
12.1	Introduction	489
12.2	Classification and framework	490
12.2.1	General classification and fundamental concepts	490
12.2.2	Objectives and properties	493
12.2.3	Assumptions and adversaries in key establishment protocols	495
12.3	Key transport based on symmetric encryption	497
12.3.1	Symmetric key transport and derivation without a server	497
12.3.2	Kerberos and related server-based protocols	500
12.4	Key agreement based on symmetric techniques	505
12.5	Key transport based on public-key encryption	506
12.5.1	Key transport using PK encryption without signatures	507
12.5.2	Protocols combining PK encryption and signatures	509
12.5.3	Hybrid key transport protocols using PK encryption	512
12.6	Key agreement based on asymmetric techniques	515
12.6.1	Diffie-Hellman and related key agreement protocols	515
12.6.2	Implicitly-certified public keys	520
12.6.3	Diffie-Hellman protocols using implicitly-certified keys	522
12.7	Secret sharing	524
12.7.1	Simple shared control schemes	524
12.7.2	Threshold schemes	525
12.7.3	Generalized secret sharing	526
12.8	Conference keying	528
12.9	Analysis of key establishment protocols	530
12.9.1	Attack strategies and classic protocol flaws	530

12.9.2 Analysis objectives and methods	532
12.10 Notes and further references	534
13 Key Management Techniques	543
13.1 Introduction	543
13.2 Background and basic concepts	544
13.2.1 Classifying keys by algorithm type and intended use	544
13.2.2 Key management objectives, threats, and policy	545
13.2.3 Simple key establishment models	546
13.2.4 Roles of third parties	547
13.2.5 Tradeoffs among key establishment protocols	550
13.3 Techniques for distributing confidential keys	551
13.3.1 Key layering and cryptoperiods	551
13.3.2 Key translation centers and symmetric-key certificates	553
13.4 Techniques for distributing public keys	555
13.4.1 Authentication trees	556
13.4.2 Public-key certificates	559
13.4.3 Identity-based systems	561
13.4.4 Implicitly-certified public keys	562
13.4.5 Comparison of techniques for distributing public keys	563
13.5 Techniques for controlling key usage	567
13.5.1 Key separation and constraints on key usage	567
13.5.2 Techniques for controlling use of symmetric keys	568
13.6 Key management involving multiple domains	570
13.6.1 Trust between two domains	570
13.6.2 Trust models involving multiple certification authorities	572
13.6.3 Certificate distribution and revocation	576
13.7 Key life cycle issues	577
13.7.1 Lifetime protection requirements	578
13.7.2 Key management life cycle	578
13.8 Advanced trusted third party services	581
13.8.1 Trusted timestamping service	581
13.8.2 Non-repudiation and notarization of digital signatures	582
13.8.3 Key escrow	584
13.9 Notes and further references	586
14 Efficient Implementation	591
14.1 Introduction	591
14.2 Multiple-precision integer arithmetic	592
14.2.1 Radix representation	592
14.2.2 Addition and subtraction	594
14.2.3 Multiplication	595
14.2.4 Squaring	596
14.2.5 Division	598
14.3 Multiple-precision modular arithmetic	599
14.3.1 Classical modular multiplication	600
14.3.2 Montgomery reduction	600
14.3.3 Barrett reduction	603
14.3.4 Reduction methods for moduli of special form	605
14.4 Greatest common divisor algorithms	606

14.4.1	Binary gcd algorithm	606
14.4.2	Lehmer's gcd algorithm	607
14.4.3	Binary extended gcd algorithm	608
14.5	Chinese remainder theorem for integers	610
14.5.1	Residue number systems	611
14.5.2	Garner's algorithm	612
14.6	Exponentiation	613
14.6.1	Techniques for general exponentiation	614
14.6.2	Fixed-exponent exponentiation algorithms	620
14.6.3	Fixed-base exponentiation algorithms	623
14.7	Exponent recoding	627
14.7.1	Signed-digit representation	627
14.7.2	String-replacement representation	628
14.8	Notes and further references	630
15	Patents and Standards	635
15.1	Introduction	635
15.2	Patents on cryptographic techniques	635
15.2.1	Five fundamental patents	636
15.2.2	Ten prominent patents	638
15.2.3	Ten selected patents	641
15.2.4	Ordering and acquiring patents	645
15.3	Cryptographic standards	645
15.3.1	International standards – cryptographic techniques	645
15.3.2	Banking security standards (ANSI, ISO)	648
15.3.3	International security architectures and frameworks	653
15.3.4	U.S. government standards (FIPS)	654
15.3.5	Internet standards and RFCs	655
15.3.6	De facto standards	656
15.3.7	Ordering and acquiring standards	656
15.4	Notes and further references	657
A	Bibliography of Papers from Selected Cryptographic Forums	663
A.1	Asiacrypt/Auscrypt Proceedings	663
A.2	Crypto Proceedings	667
A.3	Eurocrypt Proceedings	684
A.4	Fast Software-Encryption Proceedings	698
A.5	Journal of Cryptology papers	700
	References	703
	Index	755

List of Tables

1.1	Some information security objectives	3
1.2	Reference numbers comparing relative magnitudes	44
1.3	Prefixes used for various powers of 10	45
2.1	Bit complexity of basic operations in \mathbb{Z}	66
2.2	Extended Euclidean algorithm (example)	67
2.3	Orders of elements in \mathbb{Z}_{21}^*	69
2.4	Computation of $5^{596} \bmod 1234$	72
2.5	Bit complexity of basic operations in \mathbb{Z}_n	72
2.6	Jacobi symbols of elements in \mathbb{Z}_{21}^*	74
2.7	The subgroups of \mathbb{Z}_{19}^*	76
2.8	Complexity of basic operations in \mathbb{F}_{p^m}	84
2.9	The powers of x modulo $f(x) = x^4 + x + 1$	86
3.1	Some computational problems of cryptographic relevance	88
3.2	Pollard's rho algorithm (example)	107
3.3	Running time estimates for numbers factored with QS	127
4.1	Smallest strong pseudoprimes	140
4.2	Known Mersenne primes	143
4.3	Upper bounds on $p_{k,t}$ for sample values of k and t	147
4.4	Number of Miller-Rabin iterations required so that $p_{k,t} \leq (\frac{1}{2})^{80}$	148
4.5	Upper bounds on the error probability of incremental search	149
4.6	Irreducible trinomials of degree m over \mathbb{Z}_2 , $1 \leq m \leq 722$	158
4.7	Irreducible trinomials of degree m over \mathbb{Z}_2 , $723 \leq m \leq 1478$	159
4.8	Primitive polynomials over \mathbb{Z}_2	161
4.9	Primitive polynomials of degree m over \mathbb{Z}_2 , $2^m - 1$ a Mersenne prime	162
5.1	Selected percentiles of the standard normal distribution	177
5.2	Selected percentiles of the χ^2 (chi-square) distribution	178
5.3	Mean and variance of X_u for Maurer's universal statistical test	184
6.1	Berlekamp-Massey algorithm (example)	202
7.1	Estimated roughness constant κ_p for various languages	250
7.2	DES initial permutation and inverse (IP and IP ⁻¹)	253
7.3	DES per-round functions: expansion E and permutation P	253
7.4	DES key schedule bit selections (PC1 and PC2)	256
7.5	DES weak keys	258
7.6	DES pairs of semi-weak keys	258
7.7	DES strength against various attacks	259
7.8	DES S-boxes	260
7.9	FEAL functions f, f_K	261
7.10	FEAL strength against various attacks	262
7.11	IDEA decryption subkeys derived from encryption subkeys	265
7.12	IDEA encryption sample: round subkeys and ciphertext	265

7.13	IDEA decryption sample: round subkeys and variables	266
7.14	RC5 magic constants	270
8.1	Public-key encryption schemes and related computational problems . . .	284
9.1	Resistance properties required for specified data integrity applications . .	327
9.2	Design objectives for n -bit hash functions (t -bit MAC key)	335
9.3	Upper bounds on strength of selected hash functions	339
9.4	Summary of selected hash functions based on n -bit block ciphers	340
9.5	Summary of selected hash functions based on MD4	345
9.6	Test vectors for selected hash functions	345
9.7	Notation for MD4-family algorithms	345
9.8	RIPEMD-160 round function definitions	349
9.9	RIPEMD-160 word-access orders and rotate counts	351
9.10	Properties of various types of authentication	362
9.11	Definition of preimage and collision attacks	372
10.1	Bitsize of password space for various character combinations	392
10.2	Time required to search entire password space	392
10.3	Identification protocol attacks and counter-measures	418
11.1	Notation for digital signature mechanisms	427
11.2	Definition of sets and functions for modified-Rabin signatures	440
11.3	ISO/IEC 9796 notation	442
11.4	PKCS #1 notation	445
11.5	Variations of the ElGamal signing equation	457
11.6	The elements of \mathbb{F}_{2^5} as powers of a generator	459
11.7	Notation for the Rabin one-time signature scheme	463
11.8	Parameters and signatures for Merkle's one-time signature scheme . . .	467
11.9	Parameters and signatures for Merkle's one-time signature scheme . . .	467
12.1	Authentication summary – various terms and related concepts	492
12.2	Key transport protocols based on symmetric encryption	497
12.3	Selected key transport protocols based on public-key encryption	507
12.4	Selected key agreement protocols	516
12.5	Selected MTI key agreement protocols	518
13.1	Private, public, symmetric, and secret keys	544
13.2	Types of algorithms commonly used to meet specified objectives	545
13.3	Key protection requirements: symmetric-key vs. public-key systems . . .	551
14.1	Signed-magnitude and two's complement representations of integers . .	594
14.2	Multiple-precision subtraction (example)	595
14.3	Multiple-precision multiplication (example)	596
14.4	Multiple-precision squaring (example)	597
14.5	Multiple-precision division (example)	598
14.6	Multiple-precision division after normalization (example)	599
14.7	Montgomery reduction algorithm (example)	602
14.8	Montgomery multiplication (example)	603
14.9	Reduction modulo $m = b^t - c$ (example)	605
14.10	Lehmer's gcd algorithm (example)	609
14.11	Single-precision computations in Lehmer's gcd algorithm (example) . .	609

14.12	Binary extended gcd algorithm (example)	610
14.13	Inverse computation using the binary extended gcd algorithm (example)	611
14.14	Modular representations (example)	612
14.15	Sliding-window exponentiation (example)	617
14.16	Number of squarings and multiplications for exponentiation algorithms	617
14.17	Single-precision multiplications required by Montgomery exponentiation	620
14.18	Binary vector-addition chain exponentiation (example)	623
14.19	Fixed-base Euclidean method for exponentiation (example)	625
14.20	Signed-digit exponent recoding (example)	628
15.1	Five fundamental U.S. cryptographic patents	636
15.2	Ten prominent U.S. cryptographic patents	638
15.3	Ten selected U.S. cryptographic patents	641
15.4	ISO and ISO/IEC standards for generic cryptographic techniques	646
15.5	Characteristics of retail vs. wholesale banking transactions	648
15.6	ANSI encryption and banking security standards	649
15.7	ISO banking security standards	652
15.8	ISO and ISO/IEC security architectures and frameworks	653
15.9	Selected security-related U.S. FIPS Publications	654
15.10	Selected Internet RFCs	655
15.11	PKCS specifications	656

List of Figures

1.1	A taxonomy of cryptographic primitives	5
1.2	A function	7
1.3	A bijection and its inverse	8
1.4	An involution	11
1.5	A simple encryption scheme	12
1.6	Two-party communication using encryption	13
1.7	Two-party encryption with a secure channel for key exchange	16
1.8	Composition of two functions	19
1.9	Composition of two involutions	19
1.10	A signing and verification function for a digital signature scheme	22
1.11	Encryption using public-key techniques	26
1.12	Schematic use of public-key encryption	27
1.13	An impersonation attack on a two-party communication	28
1.14	A digital signature scheme with message recovery	29
1.15	Keying relationships in a simple 6-party network	36
1.16	Key management using a trusted third party (TTP)	36
1.17	Key management using public-key techniques	37
1.18	Impersonation by an active adversary	38
1.19	Authentication of public keys by a TTP	38
2.1	A functional graph	55
2.2	Conjectured relationship between some complexity classes	62
4.1	Relationships between Fermat, Euler, and strong liars	141
5.1	The normal distribution $N(0, 1)$	176
5.2	The χ^2 (chi-square) distribution with $v = 7$ degrees of freedom	177
6.1	General model of a synchronous stream cipher	193
6.2	General model of a binary additive stream cipher	194
6.3	General model of a self-synchronizing stream cipher	194
6.4	A linear feedback shift register (LFSR)	196
6.5	The LFSR $\langle 4, 1 + D + D^4 \rangle$	197
6.6	Linear complexity profile of a 20-periodic sequence	200
6.7	A feedback shift register (FSR)	202
6.8	A nonlinear combination generator	205
6.9	The Geffe generator	206
6.10	The summation generator	207
6.11	A nonlinear filter generator	208
6.12	The alternating step generator	210
6.13	The shrinking generator	211
7.1	Common modes of operation for an n -bit block cipher	229
7.2	Multiple encryption	234
7.3	The Jefferson cylinder	243

7.4	A rotor-based machine	244
7.5	Frequency of single characters in English text	247
7.6	Frequency of 15 common digrams in English text	248
7.7	Substitution-permutation (SP) network	251
7.8	DES input-output	252
7.9	DES computation path	254
7.10	DES inner function f	255
7.11	IDEA computation path	263
7.12	SAFER K-64 computation path	267
8.1	Bellare-Rogaway plaintext-aware encryption scheme	312
9.1	Simplified classification of cryptographic hash functions	324
9.2	General model for an iterated hash function	332
9.3	Three single-length, rate-one MDCs based on block ciphers	340
9.4	Compression function of MDC-2 hash function	342
9.5	Compression function of MDC-4 hash function	344
9.6	CBC-based MAC algorithm	353
9.7	The Message Authenticator Algorithm (MAA)	357
9.8	Three methods for providing data integrity using hash functions	360
10.1	Use of one-way function for password-checking	390
10.2	UNIX <i>crypt</i> password mapping	394
10.3	Functional diagram of a hand-held passcode generator	403
11.1	A taxonomy of digital signature schemes	428
11.2	Overview of a digital signature scheme with appendix	429
11.3	Overview of a digital signature scheme with message recovery	431
11.4	Signature scheme with appendix from one providing message recovery	432
11.5	Signature and verification processes for ISO/IEC 9796	443
11.6	Signature and verification processes for PKCS #1	446
11.7	An authentication tree for the Merkle one-time signature scheme	467
11.8	A full binary authentication tree of level 2 for the GMR scheme	471
12.1	Simplified classification of key establishment techniques	491
12.2	Summary of Beller-Yacobi protocol (2-pass)	515
13.1	Simple key distribution models (symmetric-key)	546
13.2	In-line, on-line, and off-line third parties	548
13.3	Third party services related to public-key certification	549
13.4	Key management: symmetric-key vs. public-key encryption	552
13.5	A binary tree	557
13.6	An authentication tree	558
13.7	Key management in different classes of asymmetric signature systems	564
13.8	Establishing trust between users in distinct domains	571
13.9	Trust models for certification	574
13.10	Key management life cycle	579
13.11	Creation and use of LEAF for key escrow data recovery	585

Foreword

by R.L. Rivest

As we draw near to closing out the twentieth century, we see quite clearly that the information-processing and telecommunications revolutions now underway will continue vigorously into the twenty-first. We interact and transact by directing flocks of digital packets towards each other through cyberspace, carrying love notes, digital cash, and secret corporate documents. Our personal and economic lives rely more and more on our ability to let such ethereal carrier pigeons mediate at a distance what we used to do with face-to-face meetings, paper documents, and a firm handshake. Unfortunately, the technical wizardry enabling remote collaborations is founded on broadcasting everything as sequences of zeros and ones that one's own dog wouldn't recognize. What is to distinguish a digital dollar when it is as easily reproducible as the spoken word? How do we converse privately when every syllable is bounced off a satellite and smeared over an entire continent? How should a bank know that it really *is* Bill Gates requesting from his laptop in Fiji a transfer of \$10,000,000,000 to another bank? Fortunately, the magical mathematics of cryptography can help. Cryptography provides techniques for keeping information secret, for determining that information has not been tampered with, and for determining who authored pieces of information.

Cryptography is fascinating because of the close ties it forges between theory and practice, and because today's practical applications of cryptography are pervasive and critical components of our information-based society. Information-protection protocols designed on theoretical foundations one year appear in products and standards documents the next. Conversely, new theoretical developments sometimes mean that last year's proposal has a previously unsuspected weakness. While the theory is advancing vigorously, there are as yet few true guarantees; the security of many proposals depends on unproven (if plausible) assumptions. The theoretical work refines and improves the practice, while the practice challenges and inspires the theoretical work. When a system is "broken," our knowledge improves, and next year's system is improved to repair the defect. (One is reminded of the long and intriguing battle between the designers of bank vaults and their opponents.)

Cryptography is also fascinating because of its game-like adversarial nature. A good cryptographer rapidly changes sides back and forth in his or her thinking, from attacker to defender and back. Just as in a game of chess, sequences of moves and counter-moves must be considered until the current situation is understood. Unlike chess players, cryptographers must also consider all the ways an adversary might try to gain by breaking the rules or violating expectations. (Does it matter if she measures how long I am computing? Does it matter if her "random" number isn't one?)

The current volume is a major contribution to the field of cryptography. It is a rigorous encyclopedia of known techniques, with an emphasis on those that are both (believed to be) secure and practically useful. It presents in a coherent manner most of the important cryptographic tools one needs to implement secure cryptographic systems, and explains many of the cryptographic principles and protocols of existing systems. The topics covered range from low-level considerations such as random-number generation and efficient modular exponentiation algorithms and medium-level items such as public-key signature techniques, to higher-level topics such as zero-knowledge protocols. This book's excellent organization and style allow it to serve well as both a self-contained tutorial and an indispensable desk reference.

In documenting the state of a fast-moving field, the authors have done incredibly well at providing error-free comprehensive content that is up-to-date. Indeed, many of the chapters, such as those on hash functions or key-establishment protocols, break new ground in both their content and their unified presentations. In the trade-off between comprehensive coverage and exhaustive treatment of individual items, the authors have chosen to write simply and directly, and thus efficiently, allowing each element to be explained together with their important details, caveats, and comparisons.

While motivated by practical applications, the authors have clearly written a book that will be of as much interest to researchers and students as it is to practitioners, by including ample discussion of the underlying mathematics and associated theoretical considerations. The essential mathematical techniques and requisite notions are presented crisply and clearly, with illustrative examples. The insightful historical notes and extensive bibliography make this book a superb stepping-stone to the literature. (I was very pleasantly surprised to find an appendix with complete programs for the CRYPTO and EUROCRYPT conferences!)

It is a pleasure to have been asked to provide the foreword for this book. I am happy to congratulate the authors on their accomplishment, and to inform the reader that he/she is looking at a landmark in the development of the field.

Ronald L. Rivest

Webster Professor of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

Preface

This book is intended as a reference for professional cryptographers, presenting the techniques and algorithms of greatest interest to the current practitioner, along with the supporting motivation and background material. It also provides a comprehensive source from which to learn cryptography, serving both students and instructors. In addition, the rigorous treatment, breadth, and extensive bibliographic material should make it an important reference for research professionals.

Our goal was to assimilate the existing cryptographic knowledge of industrial interest into one consistent, self-contained volume accessible to engineers in practice, to computer scientists and mathematicians in academia, and to motivated non-specialists with a strong desire to learn cryptography. Such a task is beyond the scope of each of the following: research papers, which by nature focus on narrow topics using very specialized (and often non-standard) terminology; survey papers, which typically address, at most, a small number of major topics at a high level; and (regretably also) most books, due to the fact that many book authors lack either practical experience or familiarity with the research literature or both. Our intent was to provide a detailed presentation of those areas of cryptography which we have found to be of greatest practical utility in our own industrial experience, while maintaining a sufficiently formal approach to be suitable both as a trustworthy reference for those whose primary interest is further research, and to provide a solid foundation for students and others first learning the subject.

Throughout each chapter, we emphasize the relationship between various aspects of cryptography. Background sections commence most chapters, providing a framework and perspective for the techniques which follow. Computer source code (e.g. C code) for algorithms has been intentionally omitted, in favor of algorithms specified in sufficient detail to allow direct implementation without consulting secondary references. We believe this style of presentation allows a better understanding of how algorithms actually work, while at the same time avoiding low-level implementation-specific constructs (which some readers will invariably be unfamiliar with) of various currently-popular programming languages.

The presentation also strongly delineates what has been established as fact (by mathematical arguments) from what is simply current conjecture. To avoid obscuring the very applied nature of the subject, rigorous proofs of correctness are in most cases omitted; however, references given in the Notes section at the end of each chapter indicate the original or recommended sources for these results. The trailing Notes sections also provide information (quite detailed in places) on various additional techniques not addressed in the main text, and provide a survey of research activities and theoretical results; references again indicate where readers may pursue particular aspects in greater depth. Needless to say, many results, and indeed some entire research areas, have been given far less attention than they warrant, or have been omitted entirely due to lack of space; we apologize in advance for such major omissions, and hope that the most significant of these are brought to our attention.

To provide an integrated treatment of cryptography spanning foundational motivation through concrete implementation, it is useful to consider a hierarchy of thought ranging from conceptual ideas and end-user services, down to the tools necessary to complete actual implementations. Table 1 depicts the hierarchical structure around which this book is organized. Corresponding to this, Figure 1 illustrates how these hierarchical levels map

...

Information Security Objectives	
Confidentiality	
Data integrity	
Authentication (entity and data origin)	
Non-repudiation	
Cryptographic functions	
Encryption	Chapters 6, 7, 8
Message authentication and data integrity techniques	Chapter 9
Identification/entity authentication techniques	Chapter 10
Digital signatures	Chapter 11
Cryptographic building blocks	
Stream ciphers	Chapter 6
Block ciphers (symmetric-key)	Chapter 7
Public-key encryption	Chapter 8
One-way hash functions (unkeyed)	Chapter 9
Message authentication codes	Chapter 9
Signature schemes (public-key, symmetric-key)	Chapter 11
Utilities	
Public-key parameter generation	Chapter 4
Pseudorandom bit generation	Chapter 5
Efficient algorithms for discrete arithmetic	Chapter 14
Foundations	
Introduction to cryptography	Chapter 1
Mathematical background	Chapter 2
Complexity and analysis of underlying problems	Chapter 3
Infrastructure techniques and commercial aspects	
Key establishment protocols	Chapter 12
Key installation and key management	Chapter 13
Cryptographic patents	Chapter 15
Cryptographic standards	Chapter 15

Table 1: Hierarchical levels of applied cryptography.

onto the various chapters, and their inter-dependence.

Table 2 lists the chapters of the book, along with the primary author(s) of each who should be contacted by readers with comments on specific chapters. Each chapter was written to provide a self-contained treatment of one major topic. Collectively, however, the chapters have been designed and carefully integrated to be entirely complementary with respect to definitions, terminology, and notation. Furthermore, there is essentially no duplication of material across chapters; instead, appropriate cross-chapter references are provided where relevant.

While it is not intended that this book be read linearly from front to back, the material has been arranged so that doing so has some merit. Two primary goals motivated by the “handbook” nature of this project were to allow easy access to stand-alone results, and to allow results and algorithms to be easily referenced (e.g., for discussion or subsequent cross-reference). To facilitate the ease of accessing and referencing results, items have been categorized and numbered to a large extent, with the following classes of items jointly numbered consecutively in each chapter: *Definitions*, *Examples*, *Facts*, *Notes*, *Remarks*, *Algorithms*, *Protocols*, and *Mechanisms*. In more traditional treatments, *Facts* are usually identified as propositions, lemmas, or theorems. We use numbered *Notes* for additional technical points,

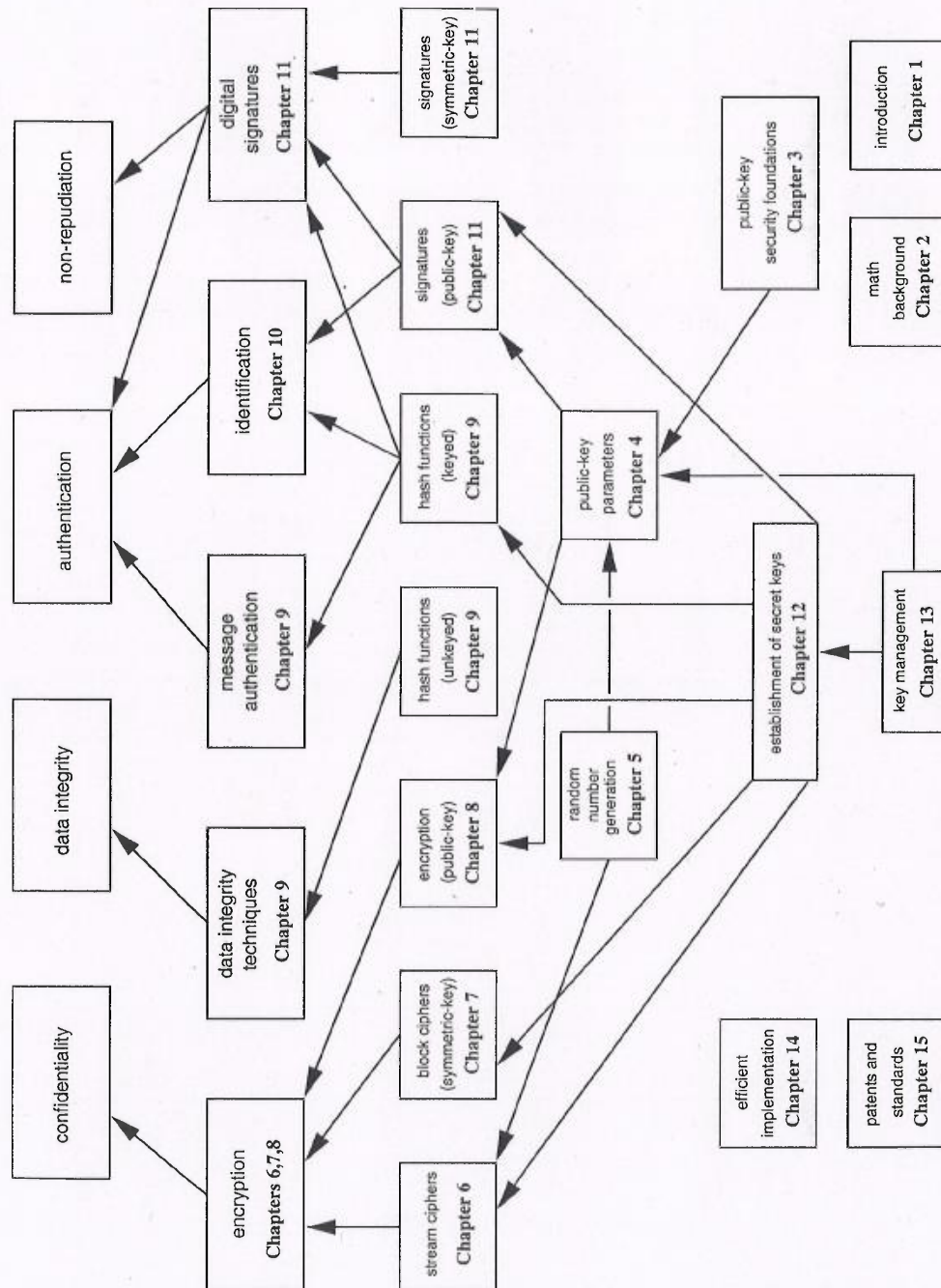


Figure 1: Roadmap of the book.

Chapter	Primary Author		
	AJM	PVO	SAV
1. Overview of Cryptography	*	*	*
2. Mathematical Background	*		
3. Number-Theoretic Reference Problems	*		
4. Public-Key Parameters	*	*	
5. Pseudorandom Bits and Sequences	*		
6. Stream Ciphers	*		
7. Block Ciphers		*	
8. Public-Key Encryption	*		
9. Hash Functions and Data Integrity		*	
10. Identification and Entity Authentication		*	
11. Digital Signatures			*
12. Key Establishment Protocols		*	
13. Key Management Techniques		*	
14. Efficient Implementation			*
15. Patents and Standards		*	
— Overall organization	*	*	

Table 2: Primary authors of each chapter.

while numbered *Remarks* identify non-technical (often non-rigorous) comments, observations, and opinions. *Algorithms*, *Protocols* and *Mechanisms* refer to techniques involving a series of steps. *Examples*, *Notes*, and *Remarks* generally begin with parenthetical summary titles to allow faster access, by indicating the nature of the content so that the entire item itself need not be read in order to determine this. The use of a large number of small subsections is also intended to enhance the handbook nature and accessibility to results.

Regarding the partitioning of subject areas into chapters, we have used what we call a *functional organization* (based on functions of interest to end-users). For example, all items related to entity authentication are addressed in one chapter. An alternative would have been what may be called an *academic organization*, under which perhaps, all protocols based on zero-knowledge concepts (including both a subset of entity authentication protocols and signature schemes) might be covered in one chapter. We believe that a functional organization is more convenient to the practitioner, who is more likely to be interested in options available for an entity authentication protocol (Chapter 10) or a signature scheme (Chapter 11), than to be seeking a zero-knowledge protocol with unspecified end-purpose.

In the front matter, a top-level Table of Contents (giving chapter numbers and titles only) is provided, as well as a detailed Table of Contents (down to the level of subsections, e.g., §5.1.1). This is followed by a List of Figures, and a List of Tables. At the start of each chapter, a brief Table of Contents (specifying section number and titles only, e.g., §5.1, §5.2) is also given for convenience.

At the end of the book, we have included a list of papers presented at each of the Crypto, Eurocrypt, Asiacrypt/Auscrypt and Fast Software Encryption conferences to date, as well as a list of all papers published in the *Journal of Cryptology* up to Volume 9. These are in addition to the *References* section, each entry of which is cited at least once in the body of the handbook. Almost all of these references have been verified for correctness in their exact titles, volume and page numbers, etc. Finally, an extensive Index prepared by the authors is included. The Index begins with a List of Symbols.

Our intention was not to introduce a collection of new techniques and protocols, but

rather to selectively present techniques from those currently available in the public domain. Such a consolidation of the literature is necessary from time to time. The fact that many good books in this field include essentially no more than what is covered here in Chapters 7, 8 and 11 (indeed, these might serve as an introductory course along with Chapter 1) illustrates that the field has grown tremendously in the past 15 years. The mathematical foundation presented in Chapters 2 and 3 is hard to find in one volume, and missing from most cryptography texts. The material in Chapter 4 on generation of public-key parameters, and in Chapter 14 on efficient implementations, while well-known to a small body of specialists and available in the scattered literature, has previously not been available in general texts. The material in Chapters 5 and 6 on pseudorandom number generation and stream ciphers is also often absent (many texts focus entirely on block ciphers), or approached only from a theoretical viewpoint. Hash functions (Chapter 9) and identification protocols (Chapter 10) have only recently been studied in depth as specialized topics on their own, and along with Chapter 12 on key establishment protocols, it is hard to find consolidated treatments of these now-mainstream topics. Key management techniques as presented in Chapter 13 have traditionally not been given much attention by cryptographers, but are of great importance in practice. A focused treatment of cryptographic patents and a concise summary of cryptographic standards, as presented in Chapter 15, are also long overdue.

In most cases (with some historical exceptions), where algorithms are known to be insecure, we have chosen to leave out specification of their details, because most such techniques are of little practical interest. Essentially all of the algorithms included have been verified for correctness by independent implementation, confirming the test vectors specified.

Acknowledgements

This project would not have been possible without the tremendous efforts put forth by our peers who have taken the time to read endless drafts and provide us with technical corrections, constructive feedback, and countless suggestions. In particular, the advice of our Advisory Editors has been invaluable, and it is impossible to attribute individual credit for their many suggestions throughout this book. Among our Advisory Editors, we would particularly like to thank:

Mihir Bellare	Don Coppersmith	Dorothy Denning	Walter Fumy
Burt Kaliski	Peter Landrock	Arjen Lenstra	Ueli Maurer
Chris Mitchell	Tatsuaki Okamoto	Bart Preneel	Ron Rivest
Gus Simmons	Miles Smid	Jacques Stern	Mike Wiener
Yacov Yacobi			

In addition, we gratefully acknowledge the exceptionally large number of additional individuals who have helped improve the quality of this volume, by providing highly appreciated feedback and guidance on various matters. These individuals include:

Carlisle Adams	Rich Ankney	Tom Berson
Simon Blackburn	Ian Blake	Antoon Bosselaers
Colin Boyd	Jörgen Brandt	Mike Burmester
Ed Dawson	Peter de Rooij	Yvo Desmedt
Whit Diffie	Hans Dobbertin	Carl Ellison
Luis Encinas	Warwick Ford	Amparo Fuster
Shuhong Gao	Will Gilbert	Marc Girault
Jovan Golić	Dieter Gollmann	Li Gong

Carrie Grant	Blake Greenlee	Helen Gustafson
Darrel Hankerson	Anwar Hasan	Don Johnson
Mike Just	Andy Klapper	Lars Knudsen
Neal Koblit	Çetin Koç	Judy Koeller
Evangelos Kranakis	David Kravitz	Hugo Krawczyk
Xuejia Lai	Charles Lam	Alan Ling
S. Mike Matyas	Willi Meier	Peter Montgomery
Mike Mosca	Tim Moses	Serge Mister
Volker Müller	David Naccache	James Nechvatal
Kaisa Nyberg	Andrew Odlyzko	Richard Outerbridge
Walter Penzhorn	Birgit Pfitzmann	Kevin Phelps
Leon Pintsov	Fred Piper	Carl Pomerance
Matt Robshaw	Peter Rodney	Phil Rogaway
Rainer Rueppel	Mahmoud Salmasizadeh	Roger Schlafly
Jeff Shallit	Jon Sorenson	Doug Stinson
Andrea Vanstone	Serge Vaudenay	Klaus Vedder
Jerry Veeh	Fausto Vitini	Lisa Yin
Robert Zuccherato		

We apologize to those whose names have inadvertently escaped this list. Special thanks are due to Carrie Grant, Darrel Hankerson, Judy Koeller, Charles Lam, and Andrea Vanstone. Their hard work contributed greatly to the quality of this book, and it was truly a pleasure working with them. Thanks also to the folks at CRC Press, including Tia Atchison, Gary Bennett, Susie Carlisle, Nora Konopka, Mary Kugler, Amy Morrell, Tim Pletscher, Bob Stern, and Wayne Yuhasz. The second author would also like to thank his colleagues past and present at Nortel Secure Networks (Bell-Northern Research), many of whom are mentioned above, for their contributions on this project, and in particular Brian O'Higgins for his encouragement and support; all views expressed, however, are entirely that of the author.

Any errors that remain are, of course, entirely our own. We would be grateful if readers who spot errors, missing references or credits, or incorrectly attributed results would contact us with details. It is our hope that this volume facilitates further advancement of the field, and that we have helped play a small part in this.

Alfred J. Menezes
Paul C. van Oorschot
Scott A. Vanstone

Chapter 1

Overview of Cryptography

Contents in Brief

1.1	Introduction	1
1.2	Information security and cryptography	2
1.3	Background on functions	6
1.4	Basic terminology and concepts	11
1.5	Symmetric-key encryption	15
1.6	Digital signatures	22
1.7	Authentication and identification	24
1.8	Public-key cryptography	25
1.9	Hash functions	33
1.10	Protocols and mechanisms	33
1.11	Key establishment, management, and certification	35
1.12	Pseudorandom numbers and sequences	39
1.13	Classes of attacks and security models	41
1.14	Notes and further references	45

1.1 Introduction

Cryptography has a long and fascinating history. The most complete non-technical account of the subject is Kahn's *The Codebreakers*. This book traces cryptography from its initial and limited use by the Egyptians some 4000 years ago, to the twentieth century where it played a crucial role in the outcome of both world wars. Completed in 1963, Kahn's book covers those aspects of the history which were most significant (up to that time) to the development of the subject. The predominant practitioners of the art were those associated with the military, the diplomatic service and government in general. Cryptography was used as a tool to protect national secrets and strategies.

The proliferation of computers and communications systems in the 1960s brought with it a demand from the private sector for means to protect information in digital form and to provide security services. Beginning with the work of Feistel at IBM in the early 1970s and culminating in 1977 with the adoption as a U.S. Federal Information Processing Standard for encrypting unclassified information, DES, the Data Encryption Standard, is the most well-known cryptographic mechanism in history. It remains the standard means for securing electronic commerce for many financial institutions around the world.

The most striking development in the history of cryptography came in 1976 when Diffie and Hellman published *New Directions in Cryptography*. This paper introduced the revolutionary concept of public-key cryptography and also provided a new and ingenious method

privacy or confidentiality	keeping information secret from all but those who are authorized to see it.
data integrity	ensuring information has not been altered by unauthorized or unknown means.
entity authentication or identification	corroboration of the identity of an entity (e.g., a person, a computer terminal, a credit card, etc.).
message authentication	corroborating the source of information; also known as data origin authentication.
signature	a means to bind information to an entity.
authorization	conveyance, to another entity, of official sanction to do or be something.
validation	a means to provide timeliness of authorization to use or manipulate information or resources.
access control	restricting access to resources to privileged entities.
certification	endorsement of information by a trusted entity.
timestamping	recording the time of creation or existence of information.
witnessing	verifying the creation or existence of information by an entity other than the creator.
receipt	acknowledgement that information has been received.
confirmation	acknowledgement that services have been provided.
ownership	a means to provide an entity with the legal right to use or transfer a resource to others.
anonymity	concealing the identity of an entity involved in some process.
non-repudiation	preventing the denial of previous commitments or actions.
revocation	retraction of certification or authorization.

Table 1.1: *Some information security objectives.*

offense to open mail for which one is not authorized. It is sometimes the case that security is achieved not through the information itself but through the physical document recording it. For example, paper currency requires special inks and material to prevent counterfeiting.

Conceptually, the way information is recorded has not changed dramatically over time. Whereas information was typically stored and transmitted on paper, much of it now resides on magnetic media and is transmitted via telecommunications systems, some wireless. What has changed dramatically is the ability to copy and alter information. One can make thousands of identical copies of a piece of information stored electronically and each is indistinguishable from the original. With information on paper, this is much more difficult. What is needed then for a society where information is mostly stored and transmitted in electronic form is a means to ensure information security which is independent of the physical medium recording or conveying it and such that the objectives of information security rely solely on digital information itself.

One of the fundamental tools used in information security is the signature. It is a building block for many other services such as non-repudiation, data origin authentication, identification, and witnessing, to mention a few. Having learned the basics in writing, an individual is taught how to produce a handwritten signature for the purpose of identification. At contract age the signature evolves to take on a very integral part of the person's identity. This signature is intended to be unique to the individual and serve as a means to identify, authorize, and validate. With electronic information the concept of a signature needs to be

2.82 Definition (*division algorithm for integers*) If a and b are integers with $b \geq 1$, then ordinary long division of a by b yields integers q (the *quotient*) and r (the *remainder*) such that

$$a = qb + r, \quad \text{where } 0 \leq r < b.$$

Moreover, q and r are unique. The remainder of the division is denoted $a \bmod b$, and the quotient is denoted $a \operatorname{div} b$.

2.83 Fact Let $a, b \in \mathbb{Z}$ with $b \neq 0$. Then $a \operatorname{div} b = \lfloor a/b \rfloor$ and $a \bmod b = a - b\lfloor a/b \rfloor$.

2.84 Example If $a = 73$, $b = 17$, then $q = 4$ and $r = 5$. Hence $73 \bmod 17 = 5$ and $73 \operatorname{div} 17 = 4$. \square

2.85 Definition An integer c is a *common divisor* of a and b if $c|a$ and $c|b$.

2.86 Definition A non-negative integer d is the *greatest common divisor* of integers a and b , denoted $d = \gcd(a, b)$, if

- (i) d is a common divisor of a and b ; and
- (ii) whenever $c|a$ and $c|b$, then $c|d$.

Equivalently, $\gcd(a, b)$ is the largest positive integer that divides both a and b , with the exception that $\gcd(0, 0) = 0$.

2.87 Example The common divisors of 12 and 18 are $\{\pm 1, \pm 2, \pm 3, \pm 6\}$, and $\gcd(12, 18) = 6$. \square

2.88 Definition A non-negative integer d is the *least common multiple* of integers a and b , denoted $d = \operatorname{lcm}(a, b)$, if

- (i) $a|d$ and $b|d$; and
- (ii) whenever $a|c$ and $b|c$, then $d|c$.

Equivalently, $\operatorname{lcm}(a, b)$ is the smallest non-negative integer divisible by both a and b .

2.89 Fact If a and b are positive integers, then $\operatorname{lcm}(a, b) = a \cdot b / \gcd(a, b)$.

2.90 Example Since $\gcd(12, 18) = 6$, it follows that $\operatorname{lcm}(12, 18) = 12 \cdot 18 / 6 = 36$. \square

2.91 Definition Two integers a and b are said to be *relatively prime* or *coprime* if $\gcd(a, b) = 1$.

2.92 Definition An integer $p \geq 2$ is said to be *prime* if its only positive divisors are 1 and p . Otherwise, p is called *composite*.

The following are some well known facts about prime numbers.

2.93 Fact If p is prime and $p|ab$, then either $p|a$ or $p|b$ (or both).

2.94 Fact There are an infinite number of prime numbers.

2.95 Fact (*prime number theorem*) Let $\pi(x)$ denote the number of prime numbers $\leq x$. Then

$$\lim_{x \rightarrow \infty} \frac{\pi(x)}{x / \ln x} = 1.$$

- (iv) (*transitivity*) If $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$, then $a \equiv c \pmod{n}$.
 (v) If $a \equiv a_1 \pmod{n}$ and $b \equiv b_1 \pmod{n}$, then $a + b \equiv a_1 + b_1 \pmod{n}$ and $ab \equiv a_1b_1 \pmod{n}$.

The *equivalence class* of an integer a is the set of all integers congruent to a modulo n . From properties (ii), (iii), and (iv) above, it can be seen that for a fixed n the relation of congruence modulo n partitions \mathbb{Z} into equivalence classes. Now, if $a = qn + r$, where $0 \leq r < n$, then $a \equiv r \pmod{n}$. Hence each integer a is congruent modulo n to a unique integer between 0 and $n - 1$, called the *least residue* of a modulo n . Thus a and r are in the same equivalence class, and so r may simply be used to represent this equivalence class.

2.113 Definition The *integers modulo n* , denoted \mathbb{Z}_n , is the set of (equivalence classes of) integers $\{0, 1, 2, \dots, n - 1\}$. Addition, subtraction, and multiplication in \mathbb{Z}_n are performed modulo n .

2.114 Example $\mathbb{Z}_{25} = \{0, 1, 2, \dots, 24\}$. In \mathbb{Z}_{25} , $13 + 16 = 4$, since $13 + 16 = 29 \equiv 4 \pmod{25}$. Similarly, $13 \cdot 16 = 8$ in \mathbb{Z}_{25} . \square

2.115 Definition Let $a \in \mathbb{Z}_n$. The *multiplicative inverse* of a modulo n is an integer $x \in \mathbb{Z}_n$ such that $ax \equiv 1 \pmod{n}$. If such an x exists, then it is unique, and a is said to be *invertible*, or a *unit*; the inverse of a is denoted by a^{-1} .

2.116 Definition Let $a, b \in \mathbb{Z}_n$. *Division* of a by b modulo n is the product of a and b^{-1} modulo n , and is only defined if b is invertible modulo n .

2.117 Fact Let $a \in \mathbb{Z}_n$. Then a is invertible if and only if $\gcd(a, n) = 1$.

2.118 Example The invertible elements in \mathbb{Z}_9 are 1, 2, 4, 5, 7, and 8. For example, $4^{-1} = 7$ because $4 \cdot 7 \equiv 1 \pmod{9}$. \square

The following is a generalization of Fact 2.117.

2.119 Fact Let $d = \gcd(a, n)$. The congruence equation $ax \equiv b \pmod{n}$ has a solution x if and only if d divides b , in which case there are exactly d solutions between 0 and $n - 1$; these solutions are all congruent modulo n/d .

2.120 Fact (*Chinese remainder theorem, CRT*) If the integers n_1, n_2, \dots, n_k are pairwise relatively prime, then the system of simultaneous congruences

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

has a unique solution modulo $n = n_1n_2 \cdots n_k$.

2.121 Algorithm (*Gauss's algorithm*) The solution x to the simultaneous congruences in the Chinese remainder theorem (Fact 2.120) may be computed as $x = \sum_{i=1}^k a_i N_i M_i \pmod{n}$, where $N_i = n/n_i$ and $M_i = N_i^{-1} \pmod{n_i}$. These computations can be performed in $O((\lg n)^2)$ bit operations.

Another efficient practical algorithm for solving simultaneous congruences in the Chinese remainder theorem is presented in §14.5.

2.122 Example The pair of congruences $x \equiv 3 \pmod{7}$, $x \equiv 7 \pmod{13}$ has a unique solution $x \equiv 59 \pmod{91}$. \square

2.123 Fact If $\gcd(n_1, n_2) = 1$, then the pair of congruences $x \equiv a \pmod{n_1}$, $x \equiv a \pmod{n_2}$ has a unique solution $x \equiv a \pmod{n_1 n_2}$.

2.124 Definition The *multiplicative group* of \mathbb{Z}_n is $\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid \gcd(a, n) = 1\}$. In particular, if n is a prime, then $\mathbb{Z}_n^* = \{a \mid 1 \leq a \leq n-1\}$.

2.125 Definition The *order* of \mathbb{Z}_n^* is defined to be the number of elements in \mathbb{Z}_n^* , namely $|\mathbb{Z}_n^*|$.

It follows from the definition of the Euler phi function (Definition 2.100) that $|\mathbb{Z}_n^*| = \phi(n)$. Note also that if $a \in \mathbb{Z}_n^*$ and $b \in \mathbb{Z}_n^*$, then $a \cdot b \in \mathbb{Z}_n^*$, and so \mathbb{Z}_n^* is closed under multiplication.

2.126 Fact Let $n \geq 2$ be an integer.

- (i) (*Euler's theorem*) If $a \in \mathbb{Z}_n^*$, then $a^{\phi(n)} \equiv 1 \pmod{n}$.
- (ii) If n is a product of distinct primes, and if $r \equiv s \pmod{\phi(n)}$, then $a^r \equiv a^s \pmod{n}$ for all integers a . In other words, when working modulo such an n , exponents can be reduced modulo $\phi(n)$.

A special case of Euler's theorem is Fermat's (little) theorem.

2.127 Fact Let p be a prime.

- (i) (*Fermat's theorem*) If $\gcd(a, p) = 1$, then $a^{p-1} \equiv 1 \pmod{p}$.
- (ii) If $r \equiv s \pmod{p-1}$, then $a^r \equiv a^s \pmod{p}$ for all integers a . In other words, when working modulo a prime p , exponents can be reduced modulo $p-1$.
- (iii) In particular, $a^p \equiv a \pmod{p}$ for all integers a .

2.128 Definition Let $a \in \mathbb{Z}_n^*$. The *order* of a , denoted $\text{ord}(a)$, is the least positive integer t such that $a^t \equiv 1 \pmod{n}$.

2.129 Fact If the order of $a \in \mathbb{Z}_n^*$ is t , and $a^s \equiv 1 \pmod{n}$, then t divides s . In particular, $t \mid \phi(n)$.

2.130 Example Let $n = 21$. Then $\mathbb{Z}_{21}^* = \{1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20\}$. Note that $\phi(21) = \phi(7)\phi(3) = 12 = |\mathbb{Z}_{21}^*|$. The orders of elements in \mathbb{Z}_{21}^* are listed in Table 2.3. \square

$a \in \mathbb{Z}_{21}^*$	1	2	4	5	8	10	11	13	16	17	19	20
order of a	1	6	3	6	2	6	6	2	3	6	6	2

Table 2.3: Orders of elements in \mathbb{Z}_{21}^* .

2.131 Definition Let $\alpha \in \mathbb{Z}_n^*$. If the order of α is $\phi(n)$, then α is said to be a *generator* or a *primitive element* of \mathbb{Z}_n^* . If \mathbb{Z}_n^* has a generator, then \mathbb{Z}_n^* is said to be *cyclic*.

2.159 Example (*Blum integer*) For the Blum integer $n = 21$, $J_n = \{1, 4, 5, 16, 17, 20\}$ and $\tilde{Q}_n = \{5, 17, 20\}$. The four square roots of $a = 4$ are 2, 5, 16, and 19, of which only 16 is also in Q_{21} . Thus 16 is the principal square root of 4 modulo 21. \square

2.160 Fact If $n = pq$ is a Blum integer, then the function $f : Q_n \rightarrow Q_n$ defined by $f(x) = x^2 \bmod n$ is a permutation. The inverse function of f is:

$$f^{-1}(x) = x^{((p-1)(q-1)+4)/8} \bmod n.$$

2.5 Abstract algebra

This section provides an overview of basic algebraic objects and their properties, for reference in the remainder of this handbook. Several of the definitions in §2.5.1 and §2.5.2 were presented earlier in §2.4.3 in the more concrete setting of the algebraic structure \mathbb{Z}_n^* .

2.161 Definition A *binary operation* $*$ on a set S is a mapping from $S \times S$ to S . That is, $*$ is a rule which assigns to each ordered pair of elements from S an element of S .

2.5.1 Groups

2.162 Definition A *group* $(G, *)$ consists of a set G with a binary operation $*$ on G satisfying the following three axioms.

- (i) The group operation is *associative*. That is, $a * (b * c) = (a * b) * c$ for all $a, b, c \in G$.
- (ii) There is an element $1 \in G$, called the *identity element*, such that $a * 1 = 1 * a = a$ for all $a \in G$.
- (iii) For each $a \in G$ there exists an element $a^{-1} \in G$, called the *inverse* of a , such that $a * a^{-1} = a^{-1} * a = 1$.

A group G is *abelian* (or *commutative*) if, furthermore,

- (iv) $a * b = b * a$ for all $a, b \in G$.

Note that multiplicative group notation has been used for the group operation. If the group operation is addition, then the group is said to be an *additive* group, the identity element is denoted by 0, and the inverse of a is denoted $-a$.

Henceforth, unless otherwise stated, the symbol $*$ will be omitted and the group operation will simply be denoted by juxtaposition.

2.163 Definition A group G is *finite* if $|G|$ is finite. The number of elements in a finite group is called its *order*.

2.164 Example The set of integers \mathbb{Z} with the operation of addition forms a group. The identity element is 0 and the inverse of an integer a is the integer $-a$. \square

2.165 Example The set \mathbb{Z}_n , with the operation of addition modulo n , forms a group of order n . The set \mathbb{Z}_n with the operation of multiplication modulo n is not a group, since not all elements have multiplicative inverses. However, the set \mathbb{Z}_n^* (see Definition 2.124) is a group of order $\phi(n)$ under the operation of multiplication modulo n , with identity element 1. \square

- 2.166 Definition** A non-empty subset H of a group G is a *subgroup* of G if H is itself a group with respect to the operation of G . If H is a subgroup of G and $H \neq G$, then H is called a *proper* subgroup of G .
- 2.167 Definition** A group G is *cyclic* if there is an element $\alpha \in G$ such that for each $b \in G$ there is an integer i with $b = \alpha^i$. Such an element α is called a *generator* of G .
- 2.168 Fact** If G is a group and $a \in G$, then the set of all powers of a forms a cyclic subgroup of G , called the subgroup *generated by* a , and denoted by $\langle a \rangle$.
- 2.169 Definition** Let G be a group and $a \in G$. The *order* of a is defined to be the least positive integer t such that $a^t = 1$, provided that such an integer exists. If such a t does not exist, then the order of a is defined to be ∞ .
- 2.170 Fact** Let G be a group, and let $a \in G$ be an element of finite order t . Then $|\langle a \rangle|$, the size of the subgroup generated by a , is equal to t .
- 2.171 Fact** (*Lagrange's theorem*) If G is a finite group and H is a subgroup of G , then $|H|$ divides $|G|$. Hence, if $a \in G$, the order of a divides $|G|$.
- 2.172 Fact** Every subgroup of a cyclic group G is also cyclic. In fact, if G is a cyclic group of order n , then for each positive divisor d of n , G contains exactly one subgroup of order d .
- 2.173 Fact** Let G be a group.
- (i) If the order of $a \in G$ is t , then the order of a^k is $t / \gcd(t, k)$.
 - (ii) If G is a cyclic group of order n and $d|n$, then G has exactly $\phi(d)$ elements of order d . In particular, G has $\phi(n)$ generators.
- 2.174 Example** Consider the multiplicative group $\mathbb{Z}_{19}^* = \{1, 2, \dots, 18\}$ of order 18. The group is cyclic (Fact 2.132(i)), and a generator is $\alpha = 2$. The subgroups of \mathbb{Z}_{19}^* , and their generators, are listed in Table 2.7. □

Subgroup	Generators	Order
$\{1\}$	1	1
$\{1, 18\}$	18	2
$\{1, 7, 11\}$	7, 11	3
$\{1, 7, 8, 11, 12, 18\}$	8, 12	6
$\{1, 4, 5, 6, 7, 9, 11, 16, 17\}$	4, 5, 6, 9, 16, 17	9
$\{1, 2, 3, \dots, 18\}$	2, 3, 10, 13, 14, 15	18

Table 2.7: The subgroups of \mathbb{Z}_{19}^* .

2.5.2 Rings

- 2.175 Definition** A *ring* $(R, +, \times)$ consists of a set R with two binary operations arbitrarily denoted $+$ (addition) and \times (multiplication) on R , satisfying the following axioms.
- (i) $(R, +)$ is an abelian group with identity denoted 0.

- (ii) The operation \times is associative. That is, $a \times (b \times c) = (a \times b) \times c$ for all $a, b, c \in R$.
- (iii) There is a multiplicative identity denoted 1 , with $1 \neq 0$, such that $1 \times a = a \times 1 = a$ for all $a \in R$.
- (iv) The operation \times is *distributive* over $+$. That is, $a \times (b + c) = (a \times b) + (a \times c)$ and $(b + c) \times a = (b \times a) + (c \times a)$ for all $a, b, c \in R$.

The ring is a *commutative ring* if $a \times b = b \times a$ for all $a, b \in R$.

2.176 Example The set of integers \mathbb{Z} with the usual operations of addition and multiplication is a commutative ring. □

2.177 Example The set \mathbb{Z}_n with addition and multiplication performed modulo n is a commutative ring. □

2.178 Definition An element a of a ring R is called a *unit* or an *invertible element* if there is an element $b \in R$ such that $a \times b = 1$.

2.179 Fact The set of units in a ring R forms a group under multiplication, called the *group of units* of R .

2.180 Example The group of units of the ring \mathbb{Z}_n is \mathbb{Z}_n^* (see Definition 2.124). □

2.5.3 Fields

2.181 Definition A *field* is a commutative ring in which all non-zero elements have multiplicative inverses.

2.182 Definition The *characteristic* of a field is 0 if $\overbrace{1 + 1 + \cdots + 1}^{m \text{ times}}$ is never equal to 0 for any $m \geq 1$. Otherwise, the characteristic of the field is the least positive integer m such that $\sum_{i=1}^m 1$ equals 0.

2.183 Example The set of integers under the usual operations of addition and multiplication is not a field, since the only non-zero integers with multiplicative inverses are 1 and -1 . However, the rational numbers \mathbb{Q} , the real numbers \mathbb{R} , and the complex numbers \mathbb{C} form fields of characteristic 0 under the usual operations. □

2.184 Fact \mathbb{Z}_n is a field (under the usual operations of addition and multiplication modulo n) if and only if n is a prime number. If n is prime, then \mathbb{Z}_n has characteristic n .

2.185 Fact If the characteristic m of a field is not 0, then m is a prime number.

2.186 Definition A subset F of a field E is a *subfield* of E if F is itself a field with respect to the operations of E . If this is the case, E is said to be an *extension field* of F .

2.209 Fact (*existence and uniqueness of finite fields*)

- (i) If F is a finite field, then F contains p^m elements for some prime p and integer $m \geq 1$.
- (ii) For every prime power order p^m , there is a unique (up to isomorphism) finite field of order p^m . This field is denoted by \mathbb{F}_{p^m} , or sometimes by $GF(p^m)$.

Informally speaking, two fields are *isomorphic* if they are structurally the same, although the representation of their field elements may be different. Note that if p is a prime then \mathbb{Z}_p is a field, and hence every field of order p is isomorphic to \mathbb{Z}_p . Unless otherwise stated, the finite field \mathbb{F}_p will henceforth be identified with \mathbb{Z}_p .

2.210 Fact If \mathbb{F}_q is a finite field of order $q = p^m$, p a prime, then the characteristic of \mathbb{F}_q is p . Moreover, \mathbb{F}_q contains a copy of \mathbb{Z}_p as a subfield. Hence \mathbb{F}_q can be viewed as an extension field of \mathbb{Z}_p of degree m .

2.211 Fact (*subfields of a finite field*) Let \mathbb{F}_q be a finite field of order $q = p^m$. Then every subfield of \mathbb{F}_q has order p^n , for some n that is a positive divisor of m . Conversely, if n is a positive divisor of m , then there is exactly one subfield of \mathbb{F}_q of order p^n ; an element $a \in \mathbb{F}_q$ is in the subfield \mathbb{F}_{p^n} if and only if $a^{p^n} = a$.

2.212 Definition The non-zero elements of \mathbb{F}_q form a group under multiplication called the *multiplicative group* of \mathbb{F}_q , denoted by \mathbb{F}_q^* .

2.213 Fact \mathbb{F}_q^* is a cyclic group of order $q - 1$. Hence $a^q = a$ for all $a \in \mathbb{F}_q$.

2.214 Definition A generator of the cyclic group \mathbb{F}_q^* is called a *primitive element* or *generator* of \mathbb{F}_q .

2.215 Fact If $a, b \in \mathbb{F}_q$, a finite field of characteristic p , then

$$(a + b)^{p^t} = a^{p^t} + b^{p^t} \text{ for all } t \geq 0.$$

2.6.2 The Euclidean algorithm for polynomials

Let \mathbb{Z}_p be the finite field of order p . The theory of greatest common divisors and the Euclidean algorithm for integers carries over in a straightforward manner to the polynomial ring $\mathbb{Z}_p[x]$ (and more generally to the polynomial ring $F[x]$, where F is any field).

2.216 Definition Let $g(x), h(x) \in \mathbb{Z}_p[x]$, where not both are 0. Then the *greatest common divisor* of $g(x)$ and $h(x)$, denoted $\gcd(g(x), h(x))$, is the monic polynomial of greatest degree in $\mathbb{Z}_p[x]$ which divides both $g(x)$ and $h(x)$. By definition, $\gcd(0, 0) = 0$.

2.217 Fact $\mathbb{Z}_p[x]$ is a *unique factorization domain*. That is, every non-zero polynomial $f(x) \in \mathbb{Z}_p[x]$ has a factorization

$$f(x) = a f_1(x)^{e_1} f_2(x)^{e_2} \cdots f_k(x)^{e_k},$$

where the $f_i(x)$ are distinct monic irreducible polynomials in $\mathbb{Z}_p[x]$, the e_i are positive integers, and $a \in \mathbb{Z}_p$. Furthermore, the factorization is unique up to rearrangement of factors.

The following is the polynomial version of the Euclidean algorithm (cf. Algorithm 2.104).

3.6 The discrete logarithm problem

The security of many cryptographic techniques depends on the intractability of the discrete logarithm problem. A partial list of these includes Diffie-Hellman key agreement and its derivatives (§12.6), ElGamal encryption (§8.4), and the ElGamal signature scheme and its variants (§11.5). This section summarizes the current knowledge regarding algorithms for solving the discrete logarithm problem.

Unless otherwise specified, algorithms in this section are described in the general setting of a (multiplicatively written) finite cyclic group G of order n with generator α (see Definition 2.167). For a more concrete approach, the reader may find it convenient to think of G as the multiplicative group \mathbb{Z}_p^* of order $p - 1$, where the group operation is simply multiplication modulo p .

3.48 Definition Let G be a finite cyclic group of order n . Let α be a generator of G , and let $\beta \in G$. The *discrete logarithm of β to the base α* , denoted $\log_\alpha \beta$, is the unique integer x , $0 \leq x \leq n - 1$, such that $\beta = \alpha^x$.

3.49 Example Let $p = 97$. Then \mathbb{Z}_{97}^* is a cyclic group of order $n = 96$. A generator of \mathbb{Z}_{97}^* is $\alpha = 5$. Since $5^{32} \equiv 35 \pmod{97}$, $\log_5 35 = 32$ in \mathbb{Z}_{97}^* . \square

The following are some elementary facts about logarithms.

3.50 Fact Let α be a generator of a cyclic group G of order n , and let $\beta, \gamma \in G$. Let s be an integer. Then $\log_\alpha(\beta\gamma) = (\log_\alpha \beta + \log_\alpha \gamma) \bmod n$ and $\log_\alpha(\beta^s) = s \log_\alpha \beta \bmod n$.

The groups of most interest in cryptography are the multiplicative group \mathbb{F}_q^* of the finite field \mathbb{F}_q (§2.6), including the particular cases of the multiplicative group \mathbb{Z}_p^* of the integers modulo a prime p , and the multiplicative group $\mathbb{F}_{2^m}^*$ of the finite field \mathbb{F}_{2^m} of characteristic two. Also of interest are the group of units \mathbb{Z}_n^* where n is a composite integer, the group of points on an elliptic curve defined over a finite field, and the jacobian of a hyperelliptic curve defined over a finite field.

3.51 Definition The *discrete logarithm problem (DLP)* is the following: given a prime p , a generator α of \mathbb{Z}_p^* , and an element $\beta \in \mathbb{Z}_p^*$, find the integer x , $0 \leq x \leq p - 2$, such that $\alpha^x \equiv \beta \pmod{p}$.

3.52 Definition The *generalized discrete logarithm problem (GDLP)* is the following: given a finite cyclic group G of order n , a generator α of G , and an element $\beta \in G$, find the integer x , $0 \leq x \leq n - 1$, such that $\alpha^x = \beta$.

The discrete logarithm problem in elliptic curve groups and in the jacobians of hyperelliptic curves are not explicitly considered in this section. The discrete logarithm problem in \mathbb{Z}_n^* is discussed further in §3.8.

3.53 Note (*difficulty of the GDLP is independent of generator*) Let α and γ be two generators of a cyclic group G of order n , and let $\beta \in G$. Let $x = \log_\alpha \beta$, $y = \log_\gamma \beta$, and $z = \log_\alpha \gamma$. Then $\alpha^x = \beta = \gamma^y = (\alpha^z)^y$. Consequently $x = zy \bmod n$, and

$$\log_\gamma \beta = (\log_\alpha \beta) (\log_\alpha \gamma)^{-1} \bmod n.$$

This means that any algorithm which computes logarithms to the base α can be used to compute logarithms to any other base γ that is also a generator of G .

3.54 Note (*generalization of GDLP*) A more general formulation of the GDLP is the following: given a finite group G and elements $\alpha, \beta \in G$, find an integer x such that $\alpha^x = \beta$, provided that such an integer exists. In this formulation, it is not required that G be a cyclic group, and, even if it is, it is not required that α be a generator of G . This problem may be harder to solve, in general, than GDLP. However, in the case where G is a cyclic group (for example if G is the multiplicative group of a finite field) and the order of α is known, it can be easily recognized whether an integer x satisfying $\alpha^x = \beta$ exists. This is because of the following fact: if G is a cyclic group, α is an element of order n in G , and $\beta \in G$, then there exists an integer x such that $\alpha^x = \beta$ if and only if $\beta^n = 1$.

3.55 Note (*solving the DLP in a cyclic group G of order n is in essence computing an isomorphism between G and \mathbb{Z}_n*) Even though any two cyclic groups of the same order are *isomorphic* (that is, they have the same structure although the elements may be written in different representations), an efficient algorithm for computing logarithms in one group does not necessarily imply an efficient algorithm for the other group. To see this, consider that every cyclic group of order n is isomorphic to the additive cyclic group \mathbb{Z}_n , i.e., the set of integers $\{0, 1, 2, \dots, n-1\}$ where the group operation is addition modulo n . Moreover, the discrete logarithm problem in the latter group, namely, the problem of finding an integer x such that $ax \equiv b \pmod{n}$ given $a, b \in \mathbb{Z}_n$, is easy as shown in the following. First note that there does not exist a solution x if $d = \gcd(a, n)$ does not divide b (Fact 2.119). Otherwise, if d divides b , the extended Euclidean algorithm (Algorithm 2.107) can be used to find integers s and t such that $as + nt = d$. Multiplying both sides of this equation by the integer b/d gives $a(sb/d) + n(tb/d) = b$. Reducing this equation modulo n yields $a(sb/d) \equiv b \pmod{n}$ and hence $x = (sb/d) \bmod n$ is the desired (and easily obtainable) solution.

The known algorithms for the DLP can be categorized as follows:

1. algorithms which work in arbitrary groups, e.g., exhaustive search (§3.6.1), the baby-step giant-step algorithm (§3.6.2), Pollard's rho algorithm (§3.6.3);
2. algorithms which work in arbitrary groups but are especially efficient if the order of the group has only small prime factors, e.g., Pohlig-Hellman algorithm (§3.6.4); and
3. the index-calculus algorithms (§3.6.5) which are efficient only in certain groups.

3.6.1 Exhaustive search

The most obvious algorithm for GDLP (Definition 3.52) is to successively compute $\alpha^0, \alpha^1, \alpha^2, \dots$ until β is obtained. This method takes $O(n)$ multiplications, where n is the order of α , and is therefore inefficient if n is large (i.e. in cases of cryptographic interest).

3.6.2 Baby-step giant-step algorithm

Let $m = \lceil \sqrt{n} \rceil$, where n is the order of α . The baby-step giant-step algorithm is a time-memory trade-off of the method of exhaustive search and is based on the following observation. If $\beta = \alpha^x$, then one can write $x = im + j$, where $0 \leq i, j < m$. Hence, $\alpha^x = \alpha^{im} \alpha^j$, which implies $\beta(\alpha^{-m})^i = \alpha^j$. This suggests the following algorithm for computing x .

3.61 Example (Pollard's rho algorithm for logarithms in a subgroup of \mathbb{Z}_{383}^*) The element $\alpha = 2$ is a generator of the subgroup G of \mathbb{Z}_{383}^* of order $n = 191$. Suppose $\beta = 228$. Partition the elements of G into three subsets according to the rule $x \in S_1$ if $x \equiv 1 \pmod{3}$, $x \in S_2$ if $x \equiv 0 \pmod{3}$, and $x \in S_3$ if $x \equiv 2 \pmod{3}$. Table 3.2 shows the values of x_i , a_i , b_i , x_{2i} , a_{2i} , and b_{2i} at the end of each iteration of step 2 of Algorithm 3.60. Note that $x_{14} = x_{28} = 144$. Finally, compute $r = b_{14} - b_{28} \bmod 191 = 125$, $r^{-1} = 125^{-1} \bmod 191 = 136$, and $r^{-1}(a_{28} - a_{14}) \bmod 191 = 110$. Hence, $\log_2 228 = 110$. \square

i	x_i	a_i	b_i	x_{2i}	a_{2i}	b_{2i}
1	228	0	1	279	0	2
2	279	0	2	184	1	4
3	92	0	4	14	1	6
4	184	1	4	256	2	7
5	205	1	5	304	3	8
6	14	1	6	121	6	18
7	28	2	6	144	12	38
8	256	2	7	235	48	152
9	152	2	8	72	48	154
10	304	3	8	14	96	118
11	372	3	9	256	97	119
12	121	6	18	304	98	120
13	12	6	19	121	5	51
14	144	12	38	144	10	104

Table 3.2: Intermediate steps of Pollard's rho algorithm in Example 3.61.

3.62 Fact Let G be a group of order n , a prime. Assume that the function $f : G \rightarrow G$ defined by equation (3.2) behaves like a random function. Then the expected running time of Pollard's rho algorithm for discrete logarithms in G is $O(\sqrt{n})$ group operations. Moreover, the algorithm requires negligible storage.

3.6.4 Pohlig-Hellman algorithm

Algorithm 3.63 for computing logarithms takes advantage of the factorization of the order n of the group G . Let $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$ be the prime factorization of n . If $x = \log_\alpha \beta$, then the approach is to determine $x_i = x \bmod p_i^{e_i}$ for $1 \leq i \leq r$, and then use Gauss's algorithm (Algorithm 2.121) to recover $x \bmod n$. Each integer x_i is determined by computing the digits $l_0, l_1, \dots, l_{e_i-1}$ in turn of its p_i -ary representation: $x_i = l_0 + l_1 p_i + \cdots + l_{e_i-1} p_i^{e_i-1}$, where $0 \leq l_j \leq p_i - 1$.

To see that the output of Algorithm 3.63 is correct, observe first that in step 2.3 the order of $\bar{\alpha}$ is q . Next, at iteration j of step 2.4, $\gamma = \alpha^{l_0 + l_1 q + \cdots + l_{j-1} q^{j-1}}$. Hence,

$$\begin{aligned}
 \bar{\beta} &= (\beta/\gamma)^{n/q^{j+1}} = (\alpha^{x-l_0-l_1 q-\cdots-l_{j-1} q^{j-1}})^{n/q^{j+1}} \\
 &= (\alpha^{n/q^{j+1}})^{x-l_0-l_1 q-\cdots-l_{j-1} q^{j-1}} \\
 &= (\alpha^{n/q^{j+1}})^{l_j q^j + \cdots + l_{e-1} q^{e-1}} \\
 &= (\alpha^{n/q})^{l_j + \cdots + l_{e-1} q^{e-1-j}} = (\bar{\alpha})^{l_j},
 \end{aligned}$$

the last equality being true because $\bar{\alpha}$ has order q . Hence, $\log_{\bar{\alpha}} \bar{\beta}$ is indeed equal to l_j .

3.63 Algorithm Pohlig-Hellman algorithm for computing discrete logarithms

INPUT: a generator α of a cyclic group G of order n , and an element $\beta \in G$.

OUTPUT: the discrete logarithm $x = \log_{\alpha} \beta$.

1. Find the prime factorization of n : $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, where $e_i \geq 1$.
2. For i from 1 to r do the following:
 (Compute $x_i = l_0 + l_1 p_i + \cdots + l_{e_i-1} p_i^{e_i-1}$, where $x_i = x \bmod p_i^{e_i}$)
 - 2.1 (Simplify the notation) Set $q \leftarrow p_i$ and $e \leftarrow e_i$.
 - 2.2 Set $\gamma \leftarrow 1$ and $l_{-1} \leftarrow 0$.
 - 2.3 Compute $\bar{\alpha} \leftarrow \alpha^{n/q}$.
 - 2.4 (Compute the l_j) For j from 0 to $e - 1$ do the following:
 Compute $\gamma \leftarrow \gamma \alpha^{l_{j-1} q^{j-1}}$ and $\bar{\beta} \leftarrow (\beta \gamma^{-1})^{n/q^{j+1}}$.
 Compute $l_j \leftarrow \log_{\bar{\alpha}} \bar{\beta}$ (e.g., using Algorithm 3.56; see Note 3.67(iii)).
 - 2.5 Set $x_i \leftarrow l_0 + l_1 q + \cdots + l_{e-1} q^{e-1}$.
3. Use Gauss's algorithm (Algorithm 2.121) to compute the integer x , $0 \leq x \leq n - 1$, such that $x \equiv x_i \pmod{p_i^{e_i}}$ for $1 \leq i \leq r$.
4. Return(x).

Example 3.64 illustrates Algorithm 3.63 with artificially small parameters.

3.64 Example (Pohlig-Hellman algorithm for logarithms in \mathbb{Z}_{251}^*) Let $p = 251$. The element $\alpha = 71$ is a generator of \mathbb{Z}_{251}^* of order $n = 250$. Consider $\beta = 210$. Then $x = \log_{71} 210$ is computed as follows.

1. The prime factorization of n is $250 = 2 \cdot 5^3$.
2. (a) (Compute $x_1 = x \bmod 2$)
 Compute $\bar{\alpha} = \alpha^{n/2} \bmod p = 250$ and $\bar{\beta} = \beta^{n/2} \bmod p = 250$. Then $x_1 = \log_{250} 250 = 1$.
 (b) (Compute $x_2 = x \bmod 5^3 = l_0 + l_1 5 + l_2 5^2$)
 - i. Compute $\bar{\alpha} = \alpha^{n/5} \bmod p = 20$.
 - ii. Compute $\gamma = 1$ and $\bar{\beta} = (\beta \gamma^{-1})^{n/5} \bmod p = 149$. Using exhaustive search,⁵ compute $l_0 = \log_{20} 149 = 2$.
 - iii. Compute $\gamma = \gamma \alpha^2 \bmod p = 21$ and $\bar{\beta} = (\beta \gamma^{-1})^{n/25} \bmod p = 113$. Using exhaustive search, compute $l_1 = \log_{20} 113 = 4$.
 - iv. Compute $\gamma = \gamma \alpha^{4 \cdot 5} \bmod p = 115$ and $\bar{\beta} = (\beta \gamma^{-1})^{(p-1)/125} \bmod p = 149$. Using exhaustive search, compute $l_2 = \log_{20} 149 = 2$.
 Hence, $x_2 = 2 + 4 \cdot 5 + 2 \cdot 5^2 = 72$.
3. Finally, solve the pair of congruences $x \equiv 1 \pmod{2}$, $x \equiv 72 \pmod{125}$ to get $x = \log_{71} 210 = 197$. \square

3.65 Fact Given the factorization of n , the running time of the Pohlig-Hellman algorithm (Algorithm 3.63) is $O(\sum_{i=1}^r e_i (\lg n + \sqrt{p_i}))$ group multiplications.

3.66 Note (effectiveness of Pohlig-Hellman) Fact 3.65 implies that the Pohlig-Hellman algorithm is efficient only if each prime divisor p_i of n is relatively small; that is, if n is a smooth

⁵Exhaustive search is preferable to Algorithm 3.56 when the group is very small (here the order of $\bar{\alpha}$ is 5).

integer (Definition 3.13). An example of a group in which the Pohlig-Hellman algorithm is effective follows. Consider the multiplicative group \mathbb{Z}_p^* where p is the 107-digit prime:

$$p = 227088231986781039743145181950291021585250524967592855 \\ 96453269189798311427475159776411276642277139650833937.$$

The order of \mathbb{Z}_p^* is $n = p - 1 = 2^4 \cdot 104729^8 \cdot 224737^8 \cdot 350377^4$. Since the largest prime divisor of $p - 1$ is only 350377, it is relatively easy to compute logarithms in this group using the Pohlig-Hellman algorithm.

3.67 Note (miscellaneous)

- (i) If n is a prime, then Algorithm 3.63 (Pohlig-Hellman) is the same as baby-step giant-step (Algorithm 3.56).
- (ii) In step 1 of Algorithm 3.63, a factoring algorithm which finds small factors first (e.g., Algorithm 3.9) should be employed; if the order n is not a smooth integer, then Algorithm 3.63 is inefficient anyway.
- (iii) The storage required for Algorithm 3.56 in step 2.4 can be eliminated by using instead Pollard's rho algorithm (Algorithm 3.60).

3.6.5 Index-calculus algorithm

The index-calculus algorithm is the most powerful method known for computing discrete logarithms. The technique employed does not apply to all groups, but when it does, it often gives a subexponential-time algorithm. The algorithm is first described in the general setting of a cyclic group G (Algorithm 3.68). Two examples are then presented to illustrate how the index-calculus algorithm works in two kinds of groups that are used in practical applications, namely \mathbb{Z}_p^* (Example 3.69) and $\mathbb{F}_{2^m}^*$ (Example 3.70).

The index-calculus algorithm requires the selection of a relatively small subset S of elements of G , called the *factor base*, in such a way that a significant fraction of elements of G can be efficiently expressed as products of elements from S . Algorithm 3.68 proceeds to precompute a database containing the logarithms of all the elements in S , and then reuses this database each time the logarithm of a particular group element is required.

The description of Algorithm 3.68 is incomplete for two reasons. Firstly, a technique for selecting the factor base S is not specified. Secondly, a method for efficiently generating relations of the form (3.5) and (3.7) is not specified. The factor base S must be a subset of G that is small (so that the system of equations to be solved in step 3 is not too large), but not too small (so that the expected number of trials to generate a relation (3.5) or (3.7) is not too large). Suitable factor bases and techniques for generating relations are known for some cyclic groups including \mathbb{Z}_p^* (see §3.6.5(i)) and $\mathbb{F}_{2^m}^*$ (see §3.6.5(ii)), and, moreover, the multiplicative group \mathbb{F}_q^* of a general finite field \mathbb{F}_q .

3.68 Algorithm Index-calculus algorithm for discrete logarithms in cyclic groups

INPUT: a generator α of a cyclic group G of order n , and an element $\beta \in G$.

OUTPUT: the discrete logarithm $y = \log_\alpha \beta$.

1. (Select a factor base S) Choose a subset $S = \{p_1, p_2, \dots, p_t\}$ of G such that a “significant proportion” of all elements in G can be efficiently expressed as a product of elements from S .
2. (Collect linear relations involving logarithms of elements in S)

Chapter 4

Public-Key Parameters

Contents in Brief

4.1	Introduction	133
4.2	Probabilistic primality tests	135
4.3	(True) Primality tests	142
4.4	Prime number generation	145
4.5	Irreducible polynomials over \mathbb{Z}_p	154
4.6	Generators and elements of high order	160
4.7	Notes and further references	165

4.1 Introduction

The efficient generation of public-key parameters is a prerequisite in public-key systems. A specific example is the requirement of a prime number p to define a finite field \mathbb{Z}_p for use in the Diffie-Hellman key agreement protocol and its derivatives (§12.6). In this case, an element of high order in \mathbb{Z}_p^* is also required. Another example is the requirement of primes p and q for an RSA modulus $n = pq$ (§8.2). In this case, the prime must be of sufficient size, and be “random” in the sense that the probability of any particular prime being selected must be sufficiently small to preclude an adversary from gaining advantage through optimizing a search strategy based on such probability. Prime numbers may be required to have certain additional properties, in order that they do not make the associated cryptosystems susceptible to specialized attacks. A third example is the requirement of an irreducible polynomial $f(x)$ of degree m over the finite field \mathbb{Z}_p for constructing the finite field \mathbb{F}_{p^m} . In this case, an element of high order in $\mathbb{F}_{p^m}^*$ is also required.

Chapter outline

The remainder of §4.1 introduces basic concepts relevant to prime number generation and summarizes some results on the distribution of prime numbers. Probabilistic primality tests, the most important of which is the Miller-Rabin test, are presented in §4.2. True primality tests by which arbitrary integers can be proven to be prime are the topic of §4.3; since these tests are generally more computationally intensive than probabilistic primality tests, they are not described in detail. §4.4 presents four algorithms for generating prime numbers, strong primes, and provable primes. §4.5 describes techniques for constructing irreducible and primitive polynomials, while §4.6 considers the production of generators and elements of high orders in groups. §4.7 concludes with chapter notes and references.

4.1.1 Generating large prime numbers naively

To motivate the organization of this chapter and introduce many of the relevant concepts, the problem of generating large prime numbers is first considered. The most natural method is to generate a random number n of appropriate size, and check if it is prime. This can be done by checking whether n is divisible by any of the prime numbers $\leq \sqrt{n}$. While more efficient methods are required in practice, to motivate further discussion consider the following approach:

1. Generate as *candidate* a random odd number n of appropriate size.
2. Test n for primality.
3. If n is composite, return to the first step.

A slight modification is to consider candidates restricted to some *search sequence* starting from n ; a trivial search sequence which may be used is $n, n+2, n+4, n+6, \dots$. Using specific search sequences may allow one to increase the expectation that a candidate is prime, and to find primes possessing certain additional desirable properties *a priori*.

In step 2, the test for primality might be either a test which *proves* that the candidate is prime (in which case the outcome of the generator is called a *provable prime*), or a test which establishes a weaker result, such as that n is “probably prime” (in which case the outcome of the generator is called a *probable prime*). In the latter case, careful consideration must be given to the exact meaning of this expression. So-called *probabilistic primality tests* typically determine, with mathematical certainty, which candidates n are composite, but do not provide a mathematical proof that n is prime in the case when such a number is declared to be “probably” so. In the latter case, however, when used properly one may often be able to draw conclusions more than adequate for the purpose at hand. For this reason, such tests are more properly called *compositeness tests* than probabilistic primality tests. True primality tests, which allow one to conclude with mathematical certainty that a number is prime, also exist, but generally require considerably greater computational resources.

While (true) primality tests can determine (with mathematical certainty) whether a typically random candidate number is prime, other techniques exist whereby candidates n are specially constructed such that it can be established by mathematical reasoning whether a candidate actually is prime. These are called *constructive prime generation* techniques.

A final distinction between different techniques for prime number generation is the use of randomness. Candidates are typically generated as a function of a random input. The technique used to judge the primality of the candidate, however, may or may not itself use random numbers. If it does not, the technique is *deterministic*, and the result is reproducible; if it does, the technique is said to be *randomized*. Both deterministic and randomized probabilistic primality tests exist.

In some cases, prime numbers are required which have additional properties. For example, to make the extraction of discrete logarithms in \mathbb{Z}_p^* resistant to an algorithm due to Pohlig and Hellman (§3.6.4), it is a requirement that $p-1$ have a large prime divisor. Thus techniques for generating public-key parameters, such as prime numbers, of special form need to be considered.

4.1.2 Distribution of prime numbers

Let $\pi(x)$ denote the number of primes in the interval $[2, x]$. The prime number theorem (Fact 2.95) states that $\pi(x) \sim \frac{x}{\ln x}$.¹ In other words, the number of primes in the interval

¹If $f(x)$ and $g(x)$ are two functions, then $f(x) \sim g(x)$ means that $\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = 1$.

$[2, x]$ is approximately equal to $\frac{x}{\ln x}$. The prime numbers are quite uniformly distributed, as the following three results illustrate.

4.1 Fact (Dirichlet theorem) If $\gcd(a, n) = 1$, then there are infinitely many primes congruent to a modulo n .

A more explicit version of Dirichlet's theorem is the following.

4.2 Fact Let $\pi(x, n, a)$ denote the number of primes in the interval $[2, x]$ which are congruent to a modulo n , where $\gcd(a, n) = 1$. Then

$$\pi(x, n, a) \sim \frac{x}{\phi(n) \ln x}.$$

In other words, the prime numbers are roughly uniformly distributed among the $\phi(n)$ congruence classes in \mathbb{Z}_n^* , for any value of n .

4.3 Fact (approximation for the n th prime number) Let p_n denote the n th prime number. Then $p_n \sim n \ln n$. More explicitly,

$$n \ln n < p_n < n(\ln n + \ln \ln n) \text{ for } n \geq 6.$$

4.2 Probabilistic primality tests

The algorithms in this section are methods by which arbitrary positive integers are tested to provide partial information regarding their primality. More specifically, probabilistic primality tests have the following framework. For each odd positive integer n , a set $W(n) \subset \mathbb{Z}_n$ is defined such that the following properties hold:

- (i) given $a \in \mathbb{Z}_n$, it can be checked in deterministic polynomial time whether $a \in W(n)$;
- (ii) if n is prime, then $W(n) = \emptyset$ (the empty set); and
- (iii) if n is composite, then $\#W(n) \geq \frac{n}{2}$.

4.4 Definition If n is composite, the elements of $W(n)$ are called *witnesses* to the compositeness of n , and the elements of the complementary set $L(n) = \mathbb{Z}_n \setminus W(n)$ are called *liars*.

A probabilistic primality test utilizes these properties of the sets $W(n)$ in the following manner. Suppose that n is an integer whose primality is to be determined. An integer $a \in \mathbb{Z}_n$ is chosen at random, and it is checked if $a \in W(n)$. The test outputs “composite” if $a \in W(n)$, and outputs “prime” if $a \notin W(n)$. If indeed $a \in W(n)$, then n is said to *fail the primality test for the base a* ; in this case, n is surely composite. If $a \notin W(n)$, then n is said to *pass the primality test for the base a* ; in this case, no conclusion with absolute certainty can be drawn about the primality of n , and the declaration “prime” may be incorrect.²

Any single execution of this test which declares “composite” establishes this with certainty. On the other hand, successive independent runs of the test all of which return the answer “prime” allow the confidence that the input is indeed prime to be increased to whatever level is desired — the cumulative probability of error is multiplicative over independent trials. If the test is run t times independently on the composite number n , the probability that n is declared “prime” all t times (i.e., the probability of error) is at most $(\frac{1}{2})^t$.

²This discussion illustrates why a probabilistic primality test is more properly called a *compositeness* test.

4.5 Definition An integer n which is believed to be prime on the basis of a probabilistic primality test is called a *probable prime*.

Two probabilistic primality tests are covered in this section: the Solovay-Strassen test (§4.2.2) and the Miller-Rabin test (§4.2.3). For historical reasons, the Fermat test is first discussed in §4.2.1; this test is not truly a probabilistic primality test since it usually fails to distinguish between prime numbers and special composite integers called Carmichael numbers.

4.2.1 Fermat's test

Fermat's theorem (Fact 2.127) asserts that if n is a prime and a is any integer, $1 \leq a \leq n-1$, then $a^{n-1} \equiv 1 \pmod{n}$. Therefore, given an integer n whose primality is under question, finding any integer a in this interval such that this equivalence is not true suffices to prove that n is composite.

4.6 Definition Let n be an odd composite integer. An integer a , $1 \leq a \leq n-1$, such that $a^{n-1} \not\equiv 1 \pmod{n}$ is called a *Fermat witness* (to compositeness) for n .

Conversely, finding an integer a between 1 and $n-1$ such that $a^{n-1} \equiv 1 \pmod{n}$ makes n appear to be a prime in the sense that it satisfies Fermat's theorem for the base a . This motivates the following definition and Algorithm 4.9.

4.7 Definition Let n be an odd composite integer and let a be an integer, $1 \leq a \leq n-1$. Then n is said to be a *pseudoprime to the base a* if $a^{n-1} \equiv 1 \pmod{n}$. The integer a is called a *Fermat liar* (to primality) for n .

4.8 Example (pseudoprime) The composite integer $n = 341 (= 11 \times 31)$ is a pseudoprime to the base 2 since $2^{340} \equiv 1 \pmod{341}$. \square

4.9 Algorithm Fermat primality test

FERMAT(n, t)

INPUT: an odd integer $n \geq 3$ and security parameter $t \geq 1$.

OUTPUT: an answer "prime" or "composite" to the question: "Is n prime?"

1. For i from 1 to t do the following:
 - 1.1 Choose a random integer a , $2 \leq a \leq n-2$.
 - 1.2 Compute $r = a^{n-1} \bmod n$ using Algorithm 2.143.
 - 1.3 If $r \neq 1$ then return("composite").
2. Return("prime").

If Algorithm 4.9 declares "composite", then n is certainly composite. On the other hand, if the algorithm declares "prime" then no proof is provided that n is indeed prime. Nonetheless, since pseudoprimes for a given base a are known to be rare, Fermat's test provides a correct answer on *most* inputs; this, however, is quite distinct from providing a correct answer most of the time (e.g., if run with different bases) on *every* input. In fact, it does not do the latter because there are (even rarer) composite numbers which are pseudoprimes to *every* base a for which $\gcd(a, n) = 1$.

4.10 Definition A Carmichael number n is a composite integer such that $a^{n-1} \equiv 1 \pmod{n}$ for all integers a which satisfy $\gcd(a, n) = 1$.

If n is a Carmichael number, then the only Fermat witnesses for n are those integers a , $1 \leq a \leq n-1$, for which $\gcd(a, n) > 1$. Thus, if the prime factors of n are all large, then with high probability the Fermat test declares that n is “prime”, even if the number of iterations t is large. This deficiency in the Fermat test is removed in the Solovay-Strassen and Miller-Rabin probabilistic primality tests by relying on criteria which are stronger than Fermat’s theorem.

This subsection is concluded with some facts about Carmichael numbers. If the prime factorization of n is known, then Fact 4.11 can be used to easily determine whether n is a Carmichael number.

4.11 Fact (*necessary and sufficient conditions for Carmichael numbers*) A composite integer n is a Carmichael number if and only if the following two conditions are satisfied:

- (i) n is square-free, i.e., n is not divisible by the square of any prime; and
- (ii) $p-1$ divides $n-1$ for every prime divisor p of n .

A consequence of Fact 4.11 is the following.

4.12 Fact Every Carmichael number is the product of at least three distinct primes.

4.13 Fact (*bounds for the number of Carmichael numbers*)

- (i) There are an infinite number of Carmichael numbers. In fact, there are more than $n^{2/7}$ Carmichael numbers in the interval $[2, n]$, once n is sufficiently large.
- (ii) The best upper bound known for $C(n)$, the number of Carmichael numbers $\leq n$, is:

$$C(n) \leq n^{1 - \{1 + o(1)\} \ln \ln \ln n / \ln \ln n} \quad \text{for } n \rightarrow \infty.$$

The smallest Carmichael number is $n = 561 = 3 \times 11 \times 17$. Carmichael numbers are relatively scarce; there are only 105212 Carmichael numbers $\leq 10^{15}$.

4.2.2 Solovay-Strassen test

The Solovay-Strassen probabilistic primality test was the first such test popularized by the advent of public-key cryptography, in particular the RSA cryptosystem. There is no longer any reason to use this test, because an alternative is available (the Miller-Rabin test) which is both more efficient and always at least as correct (see Note 4.33). Discussion is nonetheless included for historical completeness and to clarify this exact point, since many people continue to reference this test.

Recall (§2.4.5) that $\left(\frac{a}{n}\right)$ denotes the Jacobi symbol, and is equivalent to the Legendre symbol if n is prime. The Solovay-Strassen test is based on the following fact.

4.14 Fact (*Euler’s criterion*) Let n be an odd prime. Then $a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) \pmod{n}$ for all integers a which satisfy $\gcd(a, n) = 1$.

Fact 4.14 motivates the following definitions.

4.15 Definition Let n be an odd composite integer and let a be an integer, $1 \leq a \leq n-1$.

- (i) If either $\gcd(a, n) > 1$ or $a^{(n-1)/2} \not\equiv \left(\frac{a}{n}\right) \pmod{n}$, then a is called an *Euler witness* (to compositeness) for n .

- (ii) Otherwise, i.e., if $\gcd(a, n) = 1$ and $a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) \pmod{n}$, then n is said to be an *Euler pseudoprime to the base a* . (That is, n acts like a prime in that it satisfies Euler's criterion for the particular base a .) The integer a is called an *Euler liar* (to primality) for n .

4.16 Example (Euler pseudoprime) The composite integer 91 ($= 7 \times 13$) is an Euler pseudoprime to the base 9 since $9^{45} \equiv 1 \pmod{91}$ and $\left(\frac{9}{91}\right) = 1$. \square

Euler's criterion (Fact 4.14) can be used as a basis for a probabilistic primality test because of the following result.

4.17 Fact Let n be an odd composite integer. Then at most $\phi(n)/2$ of all the numbers a , $1 \leq a \leq n-1$, are Euler liars for n (Definition 4.15). Here, ϕ is the Euler phi function (Definition 2.100).

4.18 Algorithm Solovay-Strassen probabilistic primality test

SOLOVAY-STRASSEN(n, t)

INPUT: an odd integer $n \geq 3$ and security parameter $t \geq 1$.

OUTPUT: an answer "prime" or "composite" to the question: "Is n prime?"

1. For i from 1 to t do the following:
 - 1.1 Choose a random integer a , $2 \leq a \leq n-2$.
 - 1.2 Compute $r = a^{(n-1)/2} \pmod{n}$ using Algorithm 2.143.
 - 1.3 If $r \neq 1$ and $r \neq n-1$ then return("composite").
 - 1.4 Compute the Jacobi symbol $s = \left(\frac{a}{n}\right)$ using Algorithm 2.149.
 - 1.5 If $r \not\equiv s \pmod{n}$ then return("composite").
 2. Return("prime").
-

If $\gcd(a, n) = d$, then d is a divisor of $r = a^{(n-1)/2} \pmod{n}$. Hence, testing whether $r \neq 1$ is step 1.3, eliminates the necessity of testing whether $\gcd(a, n) \neq 1$. If Algorithm 4.18 declares "composite", then n is certainly composite because prime numbers do not violate Euler's criterion (Fact 4.14). Equivalently, if n is actually prime, then the algorithm always declares "prime". On the other hand, if n is actually composite, then since the bases a in step 1.1 are chosen independently during each iteration of step 1, Fact 4.17 can be used to deduce the following probability of the algorithm erroneously declaring "prime".

4.19 Fact (Solovay-Strassen error-probability bound) Let n be an odd composite integer. The probability that SOLOVAY-STRASSEN(n, t) declares n to be "prime" is less than $(\frac{1}{2})^t$.

4.2.3 Miller-Rabin test

The probabilistic primality test used most in practice is the Miller-Rabin test, also known as the *strong pseudoprime test*. The test is based on the following fact.

4.20 Fact Let n be an odd prime, and let $n-1 = 2^s r$ where r is odd. Let a be any integer such that $\gcd(a, n) = 1$. Then either $a^r \equiv 1 \pmod{n}$ or $a^{2^j r} \equiv -1 \pmod{n}$ for some j , $0 \leq j \leq s-1$.

Fact 4.20 motivates the following definitions.

4.21 Definition Let n be an odd composite integer and let $n - 1 = 2^s r$ where r is odd. Let a be an integer in the interval $[1, n - 1]$.

- (i) If $a^r \not\equiv 1 \pmod{n}$ and if $a^{2^j r} \not\equiv -1 \pmod{n}$ for all j , $0 \leq j \leq s - 1$, then a is called a *strong witness* (to compositeness) for n .
- (ii) Otherwise, i.e., if either $a^r \equiv 1 \pmod{n}$ or $a^{2^j r} \equiv -1 \pmod{n}$ for some j , $0 \leq j \leq s - 1$, then n is said to be a *strong pseudoprime to the base a* . (That is, n acts like a prime in that it satisfies Fact 4.20 for the particular base a .) The integer a is called a *strong liar* (to primality) for n .

4.22 Example (strong pseudoprime) Consider the composite integer $n = 91 (= 7 \times 13)$. Since $91 - 1 = 90 = 2 \times 45$, $s = 1$ and $r = 45$. Since $9^r = 9^{45} \equiv 1 \pmod{91}$, 91 is a strong pseudoprime to the base 9. The set of all strong liars for 91 is:

$$\{1, 9, 10, 12, 16, 17, 22, 29, 38, 53, 62, 69, 74, 75, 79, 81, 82, 90\}.$$

Notice that the number of strong liars for 91 is $18 = \phi(91)/4$, where ϕ is the Euler phi function (cf. Fact 4.23). \square

Fact 4.20 can be used as a basis for a probabilistic primality test due to the following result.

4.23 Fact If n is an odd composite integer, then at most $\frac{1}{4}$ of all the numbers a , $1 \leq a \leq n - 1$, are strong liars for n . In fact, if $n \neq 9$, the number of strong liars for n is at most $\phi(n)/4$, where ϕ is the Euler phi function (Definition 2.100).

4.24 Algorithm Miller-Rabin probabilistic primality test

MILLER-RABIN(n, t)

INPUT: an odd integer $n \geq 3$ and security parameter $t \geq 1$.

OUTPUT: an answer "prime" or "composite" to the question: "Is n prime?"

1. Write $n - 1 = 2^s r$ such that r is odd.
 2. For i from 1 to t do the following:
 - 2.1 Choose a random integer a , $2 \leq a \leq n - 2$.
 - 2.2 Compute $y = a^r \bmod n$ using Algorithm 2.143.
 - 2.3 If $y \neq 1$ and $y \neq n - 1$ then do the following:
 - $j \leftarrow 1$.
 - While $j \leq s - 1$ and $y \neq n - 1$ do the following:
 - Compute $y \leftarrow y^2 \bmod n$.
 - If $y = 1$ then return("composite").
 - $j \leftarrow j + 1$.
 - If $y \neq n - 1$ then return("composite").
 3. Return("prime").
-

Algorithm 4.24 tests whether each base a satisfies the conditions of Definition 4.21(i). In the fifth line of step 2.3, if $y = 1$, then $a^{2^j r} \equiv 1 \pmod{n}$. Since it is also the case that $a^{2^{j-1} r} \not\equiv \pm 1 \pmod{n}$, it follows from Fact 3.18 that n is composite (in fact $\gcd(a^{2^{j-1} r} - 1, n)$ is a non-trivial factor of n). In the seventh line of step 2.3, if $y \neq n - 1$, then a is a strong witness for n . If Algorithm 4.24 declares "composite", then n is certainly composite because prime numbers do not violate Fact 4.20. Equivalently, if n is actually prime, then the algorithm always declares "prime". On the other hand, if n is actually composite, then Fact 4.23 can be used to deduce the following probability of the algorithm erroneously declaring "prime".

4.25 Fact (*Miller-Rabin error-probability bound*) For any odd composite integer n , the probability that $\text{MILLER-RABIN}(n, t)$ declares n to be “prime” is less than $(\frac{1}{4})^t$.

4.26 Remark (*number of strong liars*) For most composite integers n , the number of strong liars for n is actually much smaller than the upper bound of $\phi(n)/4$ given in Fact 4.23. Consequently, the Miller-Rabin error-probability bound is much smaller than $(\frac{1}{4})^t$ for most positive integers n .

4.27 Example (*some composite integers have very few strong liars*) The only strong liars for the composite integer $n = 105 (= 3 \times 5 \times 7)$ are 1 and 104. More generally, if $k \geq 2$ and n is the product of the first k odd primes, there are only 2 strong liars for n , namely 1 and $n - 1$. \square

4.28 Remark (*fixed bases in Miller-Rabin*) If a_1 and a_2 are strong liars for n , their product $a_1 a_2$ is very likely, but not certain, to also be a strong liar for n . A strategy that is sometimes employed is to fix the bases a in the Miller-Rabin algorithm to be the first few primes (composite bases are ignored because of the preceding statement), instead of choosing them at random.

4.29 Definition Let p_1, p_2, \dots, p_t denote the first t primes. Then ψ_t is defined to be the smallest positive composite integer which is a strong pseudoprime to all the bases p_1, p_2, \dots, p_t .

The numbers ψ_t can be interpreted as follows: to determine the primality of any integer $n < \psi_t$, it is sufficient to apply the Miller-Rabin algorithm to n with the bases a being the first t prime numbers. With this choice of bases, the answer returned by Miller-Rabin is always correct. Table 4.1 gives the value of ψ_t for $1 \leq t \leq 8$.

t	ψ_t
1	2047
2	1373653
3	25326001
4	3215031751
5	2152302898747
6	3474749660383
7	341550071728321
8	341550071728321

Table 4.1: Smallest strong pseudoprimes. The table lists values of ψ_t , the smallest positive composite integer that is a strong pseudoprime to each of the first t prime bases, for $1 \leq t \leq 8$.

4.2.4 Comparison: Fermat, Solovay-Strassen, and Miller-Rabin

Fact 4.30 describes the relationships between Fermat liars, Euler liars, and strong liars (see Definitions 4.7, 4.15, and 4.21).

4.30 Fact Let n be an odd composite integer.

- (i) If a is an Euler liar for n , then it is also a Fermat liar for n .
- (ii) If a is a strong liar for n , then it is also an Euler liar for n .

4.31 Example (*Fermat, Euler, strong liars*) Consider the composite integer $n = 65 (= 5 \times 13)$. The Fermat liars for 65 are $\{1, 8, 12, 14, 18, 21, 27, 31, 34, 38, 44, 47, 51, 53, 57, 64\}$. The Euler liars for 65 are $\{1, 8, 14, 18, 47, 51, 57, 64\}$, while the strong liars for 65 are $\{1, 8, 18, 47, 57, 64\}$. \square

For a fixed composite candidate n , the situation is depicted in Figure 4.1. This set-

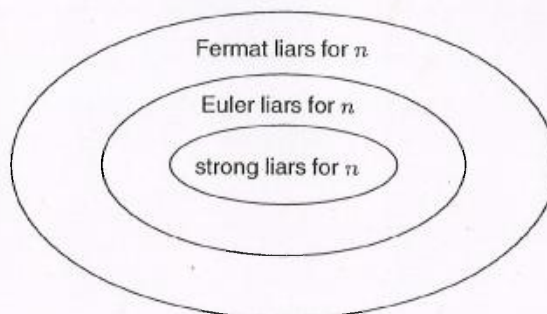


Figure 4.1: Relationships between Fermat, Euler, and strong liars for a composite integer n .

ties the question of the relative accuracy of the Fermat, Solovay-Strassen, and Miller-Rabin tests, not only in the sense of the relative correctness of each test on a fixed candidate n , but also in the sense that given n , the specified containments hold for *each* randomly chosen base a . Thus, from a correctness point of view, the Miller-Rabin test is never worse than the Solovay-Strassen test, which in turn is never worse than the Fermat test. As the following result shows, there are, however, some composite integers n for which the Solovay-Strassen and Miller-Rabin tests are equally good.

4.32 Fact If $n \equiv 3 \pmod{4}$, then a is an Euler liar for n if and only if it is a strong liar for n .

What remains is a comparison of the computational costs. While the Miller-Rabin test may appear more complex, it actually requires, at worst, the same amount of computation as Fermat's test in terms of modular multiplications; thus the Miller-Rabin test is better than Fermat's test in all regards. At worst, the sequence of computations defined in $\text{MILLER-RABIN}(n, 1)$ requires the equivalent of computing $a^{(n-1)/2} \bmod n$. It is also the case that $\text{MILLER-RABIN}(n, 1)$ requires less computation than $\text{SOLOVAY-STRASSEN}(n, 1)$, the latter requiring the computation of $a^{(n-1)/2} \bmod n$ and possibly a further Jacobi symbol computation. For this reason, the Solovay-Strassen is both computationally and conceptually more complex.

4.33 Note (*Miller-Rabin is better than Solovay-Strassen*) In summary, both the Miller-Rabin and Solovay-Strassen tests are correct in the event that either their input is actually prime, or that they declare their input composite. There is, however, no reason to use the Solovay-Strassen test (nor the Fermat test) over the Miller-Rabin test. The reasons for this are summarized below.

- (i) The Solovay-Strassen test is computationally more expensive.
- (ii) The Solovay-Strassen test is harder to implement since it also involves Jacobi symbol computations.
- (iii) The error probability for Solovay-Strassen is bounded above by $(\frac{1}{2})^t$, while the error probability for Miller-Rabin is bounded above by $(\frac{1}{4})^t$.

- (iv) Any strong liar for n is also an Euler liar for n . Hence, from a correctness point of view, the Miller-Rabin test is never worse than the Solovay-Strassen test.

4.3 (True) Primality tests

The primality tests in this section are methods by which positive integers can be *proven* to be prime, and are often referred to as *primality proving algorithms*. These primality tests are generally more computationally intensive than the probabilistic primality tests of §4.2. Consequently, before applying one of these tests to a candidate prime n , the candidate should be subjected to a probabilistic primality test such as Miller-Rabin (Algorithm 4.24).

4.34 Definition An integer n which is believed to be prime on the basis of a primality proving algorithm is called a *provable prime*.

4.3.1 Testing Mersenne numbers

Efficient algorithms are known for testing primality of some special classes of numbers, such as Mersenne numbers and Fermat numbers. Mersenne primes n are useful because the arithmetic in the field \mathbb{Z}_n for such n can be implemented very efficiently (see §14.3.4). The Lucas-Lehmer test for Mersenne numbers (Algorithm 4.37) is such an algorithm.

4.35 Definition Let $s \geq 2$ be an integer. A *Mersenne number* is an integer of the form $2^s - 1$. If $2^s - 1$ is prime, then it is called a *Mersenne prime*.

The following are necessary and sufficient conditions for a Mersenne number to be prime.

4.36 Fact Let $s \geq 3$. The Mersenne number $n = 2^s - 1$ is prime if and only if the following two conditions are satisfied:

- (i) s is prime; and
- (ii) the sequence of integers defined by $u_0 = 4$ and $u_{k+1} = (u_k^2 - 2) \bmod n$ for $k \geq 0$ satisfies $u_{s-2} = 0$.

Fact 4.36 leads to the following deterministic polynomial-time algorithm for determining (with certainty) whether a Mersenne number is prime.

4.37 Algorithm Lucas-Lehmer primality test for Mersenne numbers

INPUT: a Mersenne number $n = 2^s - 1$ with $s \geq 3$.

OUTPUT: an answer "prime" or "composite" to the question: "Is n prime?"

1. Use trial division to check if s has any factors between 2 and $\lfloor \sqrt{s} \rfloor$. If it does, then return("composite").
2. Set $u \leftarrow 4$.
3. For k from 1 to $s - 2$ do the following: compute $u \leftarrow (u^2 - 2) \bmod n$.
4. If $u = 0$ then return("prime"). Otherwise, return("composite").

It is unknown whether there are infinitely many Mersenne primes. Table 4.2 lists the 33 known Mersenne primes.

Index j	M_j	decimal digits	Index j	M_j	decimal digits
1	2	1	18	3217	969
2	3	1	19	4253	1281
3	5	2	20	4423	1332
4	7	3	21	9689	2917
5	13	4	22	9941	2993
6	17	6	23	11213	3376
7	19	6	24	19937	6002
8	31	10	25	21701	6533
9	61	19	26	23209	6987
10	89	27	27	44497	13395
11	107	33	28	86243	25962
12	127	39	29	110503	33265
13	521	157	30	132049	39751
14	607	183	31	216091	65050
15	1279	386	32?	756839	227832
16	2203	664	33?	859433	258716
17	2281	687			

Table 4.2: Known Mersenne primes. The table shows the 33 known exponents M_j , $1 \leq j \leq 33$, for which $2^{M_j} - 1$ is a Mersenne prime, and also the number of decimal digits in $2^{M_j} - 1$. The question marks after $j = 32$ and $j = 33$ indicate that it is not known whether there are any other exponents s between M_{31} and these numbers for which $2^s - 1$ is prime.

4.3.2 Primality testing using the factorization of $n - 1$

This section presents results which can be used to prove that an integer n is prime, provided that the factorization or a partial factorization of $n - 1$ is known. It may seem odd to consider a technique which requires the factorization of $n - 1$ as a subproblem — if integers of this size can be factored, the primality of n itself could be determined by factoring n . However, the factorization of $n - 1$ may be easier to compute if n has a special form, such as a *Fermat number* $n = 2^{2^k} + 1$. Another situation where the factorization of $n - 1$ may be easy to compute is when the candidate n is “constructed” by specific methods (see §4.4.4).

4.38 Fact Let $n \geq 3$ be an integer. Then n is prime if and only if there exists an integer a satisfying:

- (i) $a^{n-1} \equiv 1 \pmod{n}$; and
- (ii) $a^{(n-1)/q} \not\equiv 1 \pmod{n}$ for each prime divisor q of $n - 1$.

This result follows from the fact that \mathbb{Z}_n^* has an element of order $n - 1$ (Definition 2.128) if and only if n is prime; an element a satisfying conditions (i) and (ii) has order $n - 1$.

4.39 Note (primality test based on Fact 4.38) If n is a prime, the number of elements of order $n - 1$ is precisely $\phi(n - 1)$. Hence, to prove a candidate n prime, one simply chooses an integer $a \in \mathbb{Z}_n$ at random and uses Fact 4.38 to check if a has order $n - 1$. If this is the case, then n is certainly prime. Otherwise, another $a \in \mathbb{Z}_n$ is selected and the test is repeated. If n is indeed prime, the expected number of iterations before an element a of order $n - 1$ is selected is $O(\ln \ln n)$; this follows since $(n - 1)/\phi(n - 1) < 6 \ln \ln n$ for

$n \geq 5$ (Fact 2.102). Thus, if such an a is not found after a “reasonable” number (for example, $12 \ln \ln n$) of iterations, then n is probably composite and should again be subjected to a probabilistic primality test such as Miller-Rabin (Algorithm 4.24).³ This method is, in effect, a probabilistic compositeness test.

The next result gives a method for proving primality which requires knowledge of only a *partial* factorization of $n - 1$.

4.40 Fact (Pocklington’s theorem) Let $n \geq 3$ be an integer, and let $n = RF + 1$ (i.e. F divides $n - 1$) where the prime factorization of F is $F = \prod_{j=1}^t q_j^{e_j}$. If there exists an integer a satisfying:

- (i) $a^{n-1} \equiv 1 \pmod{n}$; and
- (ii) $\gcd(a^{(n-1)/q_j} - 1, n) = 1$ for each j , $1 \leq j \leq t$,

then every prime divisor p of n is congruent to 1 modulo F . It follows that if $F > \sqrt{n} - 1$, then n is prime.

If n is indeed prime, then the following result establishes that most integers a satisfy conditions (i) and (ii) of Fact 4.40, provided that the prime divisors of $F > \sqrt{n} - 1$ are sufficiently large.

4.41 Fact Let $n = RF + 1$ be an odd prime with $F > \sqrt{n} - 1$ and $\gcd(R, F) = 1$. Let the distinct prime factors of F be q_1, q_2, \dots, q_t . Then the probability that a randomly selected base a , $1 \leq a \leq n - 1$, satisfies both: (i) $a^{n-1} \equiv 1 \pmod{n}$; and (ii) $\gcd(a^{(n-1)/q_j} - 1, n) = 1$ for each j , $1 \leq j \leq t$, is $\prod_{j=1}^t (1 - 1/q_j) \geq 1 - \sum_{j=1}^t 1/q_j$.

Thus, if the factorization of a divisor $F > \sqrt{n} - 1$ of $n - 1$ is known, one simply chooses random integers a in the interval $[2, n - 2]$ until one is found satisfying conditions (i) and (ii) of Fact 4.40, implying that n is prime. If such an a is not found after a “reasonable” number of iterations,⁴ then n is probably composite and should be subjected to a probabilistic primality test (footnote 3 also applies here). This method is, in effect, a probabilistic compositeness test.

The next result gives a method for proving primality which only requires the factorization of a divisor F of $n - 1$ that is greater than $\sqrt[3]{n}$. For an example of the use of Fact 4.42, see Note 4.63.

4.42 Fact Let $n \geq 3$ be an odd integer. Let $n = 2RF + 1$, and suppose that there exists an integer a satisfying both: (i) $a^{n-1} \equiv 1 \pmod{n}$; and (ii) $\gcd(a^{(n-1)/q} - 1, n) = 1$ for each prime divisor q of F . Let $x \geq 0$ and y be defined by $2R = xF + y$ and $0 \leq y < F$. If $F \geq \sqrt[3]{n}$ and if $y^2 - 4x$ is neither 0 nor a perfect square, then n is prime.

4.3.3 Jacobi sum test

The *Jacobi sum test* is another true primality test. The basic idea is to test a set of congruences which are analogues of Fermat’s theorem (Fact 2.127(i)) in certain *cyclotomic rings*. The running time of the Jacobi sum test for determining the primality of an integer n is $O((\ln n)^{c \ln \ln \ln n})$ bit operations for some constant c . This is “almost” a polynomial-time algorithm since the exponent $\ln \ln \ln n$ acts like a constant for the range of values for

³ Another approach is to run both algorithms in parallel (with an unlimited number of iterations), until one of them stops with a definite conclusion “prime” or “composite”.

⁴ The number of iterations may be taken to be T where $P^T \leq (\frac{1}{2})^{100}$, and where $P = 1 - \prod_{j=1}^t (1 - 1/q_j)$.

n of interest. For example, if $n \leq 2^{512}$, then $\ln \ln \ln n < 1.78$. The version of the Jacobi sum primality test used in practice is a randomized algorithm which terminates within $O(k(\ln n)^{c \ln \ln \ln n})$ steps with probability at least $1 - (\frac{1}{2})^k$ for every $k \geq 1$, and always gives a correct answer. One drawback of the algorithm is that it does not produce a “certificate” which would enable the answer to be verified in much shorter time than running the algorithm itself.

The Jacobi sum test is, indeed, practical in the sense that the primality of numbers that are several hundred decimal digits long can be handled in just a few minutes on a computer. However, the test is not as easy to program as the probabilistic Miller-Rabin test (Algorithm 4.24), and the resulting code is not as compact. The details of the algorithm are complicated and are not given here; pointers to the literature are given in the chapter notes on page 166.

4.3.4 Tests using elliptic curves

Elliptic curve primality proving algorithms are based on an elliptic curve analogue of Pocklington’s theorem (Fact 4.40). The version of the algorithm used in practice is usually referred to as *Atkin’s test* or the *Elliptic Curve Primality Proving algorithm* (ECPP). Under heuristic arguments, the expected running time of this algorithm for proving the primality of an integer n has been shown to be $O((\ln n)^{6+\epsilon})$ bit operations for any $\epsilon > 0$. Atkin’s test has the advantage over the Jacobi sum test (§4.3.3) that it produces a short *certificate of primality* which can be used to efficiently verify the primality of the number. Atkin’s test has been used to prove the primality of numbers more than 1000 decimal digits long.

The details of the algorithm are complicated and are not presented here; pointers to the literature are given in the chapter notes on page 166.

4.4 Prime number generation

This section considers algorithms for the generation of prime numbers for cryptographic purposes. Four algorithms are presented: Algorithm 4.44 for generating *probable* primes (see Definition 4.5), Algorithm 4.53 for generating *strong* primes (see Definition 4.52), Algorithm 4.56 for generating *probable* primes p and q suitable for use in the Digital Signature Algorithm (DSA), and Algorithm 4.62 for generating *provable* primes (see Definition 4.34).

4.43 Note (*prime generation vs. primality testing*) Prime number *generation* differs from *primality testing* as described in §4.2 and §4.3, but may and typically does involve the latter. The former allows the construction of candidates of a fixed form which may lead to more efficient testing than possible for random candidates.

4.4.1 Random search for probable primes

By the prime number theorem (Fact 2.95), the proportion of (positive) integers $\leq x$ that are prime is approximately $1/\ln x$. Since half of all integers $\leq x$ are even, the proportion of *odd* integers $\leq x$ that are prime is approximately $2/\ln x$. For instance, the proportion of all odd integers $\leq 2^{512}$ that are prime is approximately $2/(512 \cdot \ln(2)) \approx 1/177$. This suggests that a reasonable strategy for selecting a random k -bit (probable) prime is to repeatedly pick random k -bit odd integers n until one is found that is declared to be “prime”

by $\text{MILLER-RABIN}(n,t)$ (Algorithm 4.24) for an appropriate value of the security parameter t (discussed below).

If a random k -bit odd integer n is divisible by a small prime, it is less computationally expensive to rule out the candidate n by trial division than by using the Miller-Rabin test. Since the probability that a random integer n has a small prime divisor is relatively large, before applying the Miller-Rabin test, the candidate n should be tested for small divisors below a pre-determined bound B . This can be done by dividing n by all the primes below B , or by computing greatest common divisors of n and (pre-computed) products of several of the primes $\leq B$. The proportion of candidate odd integers n not ruled out by this trial division is $\prod_{3 \leq p \leq B} (1 - \frac{1}{p})$ which, by Merten's theorem, is approximately $1.12 / \ln B$ (here p ranges over prime values). For example, if $B = 256$, then only 20% of candidate odd integers n pass the trial division stage, i.e., 80% are discarded before the more costly Miller-Rabin test is performed.

4.44 Algorithm Random search for a prime using the Miller-Rabin test

$\text{RANDOM-SEARCH}(k,t)$

INPUT: the required bitlength k of the prime, and a security parameter t . (See Note 4.49 for guidance on selecting t .)

OUTPUT: a random k -bit probable prime.

1. Generate an odd k -bit integer n at random.
 2. Use trial division to determine whether n is divisible by any odd prime $\leq B$ (see Note 4.45 for guidance on selecting B). If it is then go to step 1.
 3. If $\text{MILLER-RABIN}(n,t)$ (Algorithm 4.24) outputs "prime" then return(n). Otherwise, go to step 1.
-

4.45 Note (*optimal trial division bound* B) Let E denote the time for a full k -bit modular exponentiation, and let D denote the time required for ruling out one small prime as divisor of a k -bit integer. (The values E and D depend on the particular implementation of long-integer arithmetic.) Then the trial division bound B that minimizes the expected running time of Algorithm 4.44 for generating a k -bit prime is roughly $B = E/D$. A more accurate estimate of the optimum choice for B can be obtained experimentally. The odd primes up to B can be precomputed and stored in a table. If memory is scarce, a value of B that is smaller than the optimum value may be used.

Since the Miller-Rabin test does not provide a mathematical proof that a number is indeed prime, the number n returned by Algorithm 4.44 is a probable prime (Definition 4.5). It is important, therefore, to have an estimate of the probability that n is in fact composite.

4.46 Definition The probability that $\text{RANDOM-SEARCH}(k,t)$ (Algorithm 4.44) returns a composite number is denoted by $p_{k,t}$.

4.47 Note (*remarks on estimating $p_{k,t}$*) It is tempting to conclude directly from Fact 4.25 that $p_{k,t} \leq (\frac{1}{4})^t$. This reasoning is flawed (although typically the conclusion will be correct in practice) since it does not take into account the distribution of the primes. (For example, if the candidate n were chosen from a set S of composite numbers, the probability of error is 1.) The following discussion elaborates on this point. Let X represent the event that n is composite, and let Y_t denote the event that $\text{MILLER-RABIN}(n,t)$ declares n to be prime. Then Fact 4.25 states that $P(Y_t|X) \leq (\frac{1}{4})^t$. What is relevant, however, to the estimation of $p_{k,t}$ is the quantity $P(X|Y_t)$. Suppose that candidates n are drawn uniformly and randomly

from a set S of odd numbers, and suppose p is the probability that n is prime (this depends on the candidate set S). Assume also that $0 < p < 1$. Then by Bayes' theorem (Fact 2.10):

$$P(X|Y_t) = \frac{P(X)P(Y_t|X)}{P(Y_t)} \leq \frac{P(Y_t|X)}{P(Y_t)} \leq \frac{1}{p} \left(\frac{1}{4}\right)^t,$$

since $P(Y_t) \geq p$. Thus the probability $P(X|Y_t)$ may be considerably larger than $(\frac{1}{4})^t$ if p is small. However, the error-probability of Miller-Rabin is usually far smaller than $(\frac{1}{4})^t$ (see Remark 4.26). Using better estimates for $P(Y_t|X)$ and estimates on the number of k -bit prime numbers, it has been shown that $p_{k,t}$ is, in fact, smaller than $(\frac{1}{4})^t$ for all sufficiently large k . A more concrete result is the following: if candidates n are chosen at random from the set of odd numbers in the interval $[3, x]$, then $P(X|Y_t) \leq (\frac{1}{4})^t$ for all $x \geq 10^{60}$.

Further refinements for $P(Y_t|X)$ allow the following explicit upper bounds on $p_{k,t}$ for various values of k and t .⁵

4.48 Fact (some upper bounds on $p_{k,t}$ in Algorithm 4.44)

- (i) $p_{k,1} < k^2 4^{2-\sqrt{k}}$ for $k \geq 2$.
- (ii) $p_{k,t} < k^{3/2} 2^{t-1/2} 4^{2-\sqrt{tk}}$ for $(t=2, k \geq 88)$ or $(3 \leq t \leq k/9, k \geq 21)$.
- (iii) $p_{k,t} < \frac{7}{20} k 2^{-5t} + \frac{1}{7} k^{15/4} 2^{-k/2-2t} + 12k 2^{-k/4-3t}$ for $k/9 \leq t \leq k/4, k \geq 21$.
- (iv) $p_{k,t} < \frac{1}{7} k^{15/4} 2^{-k/2-2t}$ for $t \geq k/4, k \geq 21$.

For example, if $k = 512$ and $t = 6$, then Fact 4.48(ii) gives $p_{512,6} \leq (\frac{1}{2})^{88}$. In other words, the probability that RANDOM-SEARCH(512,6) returns a 512-bit composite integer is less than $(\frac{1}{2})^{88}$. Using more advanced techniques, the upper bounds on $p_{k,t}$ given by Fact 4.48 have been improved. These upper bounds arise from complicated formulae which are not given here. Table 4.3 lists some improved upper bounds on $p_{k,t}$ for some sample values of k and t . As an example, the probability that RANDOM-SEARCH(500,6) returns a composite number is $\leq (\frac{1}{2})^{92}$. Notice that the values of $p_{k,t}$ implied by the table are considerably smaller than $(\frac{1}{4})^t = (\frac{1}{2})^{2t}$.

k	t									
	1	2	3	4	5	6	7	8	9	10
100	5	14	20	25	29	33	36	39	41	44
150	8	20	28	34	39	43	47	51	54	57
200	11	25	34	41	47	52	57	61	65	69
250	14	29	39	47	54	60	65	70	75	79
300	19	33	44	53	60	67	73	78	83	88
350	28	38	48	58	66	73	80	86	91	97
400	37	46	55	63	72	80	87	93	99	105
450	46	54	62	70	78	85	93	100	106	112
500	56	63	70	78	85	92	99	106	113	119
550	65	72	79	86	93	100	107	113	119	126
600	75	82	88	95	102	108	115	121	127	133

Table 4.3: Upper bounds on $p_{k,t}$ for sample values of k and t . An entry j corresponding to k and t implies $p_{k,t} \leq (\frac{1}{2})^j$.

⁵The estimates of $p_{k,t}$ presented in the remainder of this subsection were derived for the situation where Algorithm 4.44 does not use trial division by small primes to rule out some candidates n . Since trial division never rules out a prime, it can only give a better chance of rejecting composites. Thus the error probability $p_{k,t}$ might actually be even smaller than the estimates given here.

4.49 Note (*controlling the error probability*) In practice, one is usually willing to tolerate an error probability of $(\frac{1}{2})^{80}$ when using Algorithm 4.44 to generate probable primes. For sample values of k , Table 4.4 lists the smallest value of t that can be derived from Fact 4.48 for which $p_{k,t} \leq (\frac{1}{2})^{80}$. For example, when generating 1000-bit probable primes, Miller-Rabin with $t = 3$ repetitions suffices. Algorithm 4.44 rules out most candidates n either by trial division (in step 2) or by performing just one iteration of the Miller-Rabin test (in step 3). For this reason, the only effect of selecting a larger security parameter t on the running time of the algorithm will likely be to increase the time required in the final stage when the (probable) prime is chosen.

k	t	k	t	k	t	k	t	k	t
100	27	500	6	900	3	1300	2	1700	2
150	18	550	5	950	3	1350	2	1750	2
200	15	600	5	1000	3	1400	2	1800	2
250	12	650	4	1050	3	1450	2	1850	2
300	9	700	4	1100	3	1500	2	1900	2
350	8	750	4	1150	3	1550	2	1950	2
400	7	800	4	1200	3	1600	2	2000	2
450	6	850	3	1250	3	1650	2	2050	2

Table 4.4: For sample k , the smallest t from Fact 4.48 is given for which $p_{k,t} \leq (\frac{1}{2})^{80}$.

4.50 Remark (*Miller-Rabin test with base $a = 2$*) The Miller-Rabin test involves exponentiating the base a ; this may be performed using the repeated square-and-multiply algorithm (Algorithm 2.143). If $a = 2$, then multiplication by a is a simple procedure relative to multiplying by a in general. One optimization of Algorithm 4.44 is, therefore, to fix the base $a = 2$ when first performing the Miller-Rabin test in step 3. Since most composite numbers will fail the Miller-Rabin test with base $a = 2$, this modification will lower the expected running time of Algorithm 4.44.

4.51 Note (*incremental search*)

- (i) An alternative technique to generating candidates n at random in step 1 of Algorithm 4.44 is to first select a random k -bit odd number n_0 , and then test the s numbers $n = n_0, n_0 + 2, n_0 + 4, \dots, n_0 + 2(s-1)$ for primality. If all these s candidates are found to be composite, the algorithm is said to have *failed*. If $s = c \cdot \ln 2^k$ where c is a constant, the probability $q_{k,t,s}$ that this incremental search variant of Algorithm 4.44 returns a composite number has been shown to be less than $\delta k^3 2^{-\sqrt{k}}$ for some constant δ . Table 4.5 gives some explicit bounds on this error probability for $k = 500$ and $t \leq 10$. Under reasonable number-theoretic assumptions, the probability of the algorithm failing has been shown to be less than $2e^{-2c}$ for large k (here, $e \approx 2.71828$).
- (ii) Incremental search has the advantage that fewer random bits are required. Furthermore, the trial division by small primes in step 2 of Algorithm 4.44 can be accomplished very efficiently as follows. First the values $R[p] = n_0 \bmod p$ are computed for each odd prime $p \leq B$. Each time 2 is added to the current candidate, the values in the table R are updated as $R[p] \leftarrow (R[p] + 2) \bmod p$. The candidate passes the trial division stage if and only if none of the $R[p]$ values equal 0.
- (iii) If B is large, an alternative method for doing the trial division is to initialize a table $S[i] \leftarrow 0$ for $0 \leq i \leq (s-1)$; the entry $S[i]$ corresponds to the candidate $n_0 + 2i$. For each odd prime $p \leq B$, $n_0 \bmod p$ is computed. Let j be the smallest index for

c	t									
	1	2	3	4	5	6	7	8	9	10
1	17	37	51	63	72	81	89	96	103	110
5	13	32	46	58	68	77	85	92	99	105
10	11	30	44	56	66	75	83	90	97	103

Table 4.5: Upper bounds on the error probability of incremental search (Note 4.51) for $k = 500$ and sample values of c and t . An entry j corresponding to c and t implies $q_{500,t,s} \leq (\frac{1}{2})^j$, where $s = c \cdot \ln 2^{500}$.

which $(n_0 + 2j) \equiv 0 \pmod{p}$. Then $S[j]$ and each p^{th} entry after it are set to 1. A candidate $n_0 + 2i$ then passes the trial division stage if and only if $S[i] = 0$. Note that the estimate for the optimal trial division bound B given in Note 4.45 does not apply here (nor in (ii)) since the cost of division is amortized over all candidates.

4.4.2 Strong primes

The RSA cryptosystem (§8.2) uses a modulus of the form $n = pq$, where p and q are distinct odd primes. The primes p and q must be of sufficient size that factorization of their product is beyond computational reach. Moreover, they should be random primes in the sense that they be chosen as a function of a random input through a process defining a pool of candidates of sufficient cardinality that an exhaustive attack is infeasible. In practice, the resulting primes must also be of a pre-determined bitlength, to meet system specifications. The discovery of the RSA cryptosystem led to the consideration of several additional constraints on the choice of p and q which are necessary to ensure the resulting RSA system safe from cryptanalytic attack, and the notion of a strong prime (Definition 4.52) was defined. These attacks are described at length in Note 8.8(iii); as noted there, it is now believed that strong primes offer little protection beyond that offered by random primes, since randomly selected primes of the sizes typically used in RSA moduli today will satisfy the constraints with high probability. On the other hand, they are no less secure, and require only minimal additional running time to compute; thus, there is little real additional cost in using them.

4.52 Definition A prime number p is said to be a *strong prime* if integers r , s , and t exist such that the following three conditions are satisfied:

- (i) $p - 1$ has a large prime factor, denoted r ;
- (ii) $p + 1$ has a large prime factor, denoted s ; and
- (iii) $r - 1$ has a large prime factor, denoted t .

In Definition 4.52, a precise qualification of “large” depends on specific attacks that should be guarded against; for further details, see Note 8.8(iii).

4.53 Algorithm Gordon's algorithm for generating a strong prime

SUMMARY: a strong prime p is generated.

1. Generate two large random primes s and t of roughly equal bitlength (see Note 4.54).
2. Select an integer i_0 . Find the first prime in the sequence $2it + 1$, for $i = i_0, i_0 + 1, i_0 + 2, \dots$ (see Note 4.54). Denote this prime by $r = 2it + 1$.
3. Compute $p_0 = (2s^{r-2} \bmod r)s - 1$.
4. Select an integer j_0 . Find the first prime in the sequence $p_0 + 2jrs$, for $j = j_0, j_0 + 1, j_0 + 2, \dots$ (see Note 4.54). Denote this prime by $p = p_0 + 2jrs$.
5. Return(p).

Justification. To see that the prime p returned by Gordon's algorithm is indeed a strong prime, observe first (assuming $r \neq s$) that $s^{r-1} \equiv 1 \pmod{r}$; this follows from Fermat's theorem (Fact 2.127). Hence, $p_0 \equiv 1 \pmod{r}$ and $p_0 \equiv -1 \pmod{s}$. Finally (cf. Definition 4.52),

- (i) $p - 1 = p_0 + 2jrs - 1 \equiv 0 \pmod{r}$, and hence $p - 1$ has the prime factor r ;
- (ii) $p + 1 = p_0 + 2jrs + 1 \equiv 0 \pmod{s}$, and hence $p + 1$ has the prime factor s ; and
- (iii) $r - 1 = 2it \equiv 0 \pmod{t}$, and hence $r - 1$ has the prime factor t .

4.54 Note (implementing Gordon's algorithm)

- (i) The primes s and t required in step 1 can be probable primes generated by Algorithm 4.44. The Miller-Rabin test (Algorithm 4.24) can be used to test each candidate for primality in steps 2 and 4, after ruling out candidates that are divisible by a small prime less than some bound B . See Note 4.45 for guidance on selecting B . Since the Miller-Rabin test is a probabilistic primality test, the output of this implementation of Gordon's algorithm is a probable prime.
- (ii) By carefully choosing the sizes of primes s, t and parameters i_0, j_0 , one can control the exact bitlength of the resulting prime p . Note that the bitlengths of r and s will be about half that of p , while the bitlength of t will be slightly less than that of r .

4.55 Fact (running time of Gordon's algorithm) If the Miller-Rabin test is the primality test used in steps 1, 2, and 4, the expected time Gordon's algorithm takes to find a strong prime is only about 19% more than the expected time Algorithm 4.44 takes to find a random prime.

4.4.3 NIST method for generating DSA primes

Some public-key schemes require primes satisfying various specific conditions. For example, the NIST Digital Signature Algorithm (DSA of §11.5.1) requires two primes p and q satisfying the following three conditions:

- (i) $2^{159} < q < 2^{160}$; that is, q is a 160-bit prime;
- (ii) $2^{L-1} < p < 2^L$ for a specified L , where $L = 512 + 64l$ for some $0 \leq l \leq 8$; and
- (iii) q divides $p - 1$.

This section presents an algorithm for generating such primes p and q . In the following, H denotes the SHA-1 hash function (§9.53) which maps bitstrings of bitlength $< 2^{64}$ to 160-bit hash-codes. Where required, an integer x in the range $0 \leq x < 2^g$ whose binary representation is $x = x_{g-1}2^{g-1} + x_{g-2}2^{g-2} + \dots + x_22^2 + x_12 + x_0$ should be converted to the g -bit sequence $(x_{g-1}x_{g-2} \dots x_2x_1x_0)$, and vice versa.

4.56 Algorithm NIST method for generating DSA primes

INPUT: an integer l , $0 \leq l \leq 8$.

OUTPUT: a 160-bit prime q and an L -bit prime p , where $L = 512 + 64l$ and $q|(p-1)$.

1. Compute $L = 512 + 64l$. Using long division of $(L-1)$ by 160, find n, b such that $L-1 = 160n + b$, where $0 \leq b < 160$.
2. Repeat the following:
 - 2.1 Choose a random seed s (not necessarily secret) of bitlength $g \geq 160$.
 - 2.2 Compute $U = H(s) \oplus H((s+1) \bmod 2^g)$.
 - 2.3 Form q from U by setting to 1 the most significant and least significant bits of U . (Note that q is a 160-bit odd integer.)
 - 2.4 Test q for primality using MILLER-RABIN(q, t) for $t \geq 18$ (see Note 4.57).
Until q is found to be a (probable) prime.
3. Set $i \leftarrow 0, j \leftarrow 2$.
4. While $i < 4096$ do the following:
 - 4.1 For k from 0 to n do the following: set $V_k \leftarrow H((s+j+k) \bmod 2^g)$.
 - 4.2 For the integer W defined below, let $X = W + 2^{L-1}$. (X is an L -bit integer.)

$$W = V_0 + V_1 2^{160} + V_2 2^{320} + \dots + V_{n-1} 2^{160(n-1)} + (V_n \bmod 2^b) 2^{160n}.$$
 - 4.3 Compute $c = X \bmod 2q$ and set $p = X - (c-1)$. (Note that $p \equiv 1 \pmod{2q}$.)
 - 4.4 If $p \geq 2^{L-1}$ then do the following:
 Test p for primality using MILLER-RABIN(p, t) for $t \geq 5$ (see Note 4.57).
 If p is a (probable) prime then return(q, p).
 - 4.5 Set $i \leftarrow i + 1, j \leftarrow j + n + 1$.
5. Go to step 2.

4.57 Note (choice of primality test in Algorithm 4.56)

- (i) The FIPS 186 document where Algorithm 4.56 was originally described only specifies that a *robust* primality test be used in steps 2.4 and 4.4, i.e., a primality test where the probability of a composite integer being declared prime is at most $(\frac{1}{2})^{80}$. If the heuristic assumption is made that q is a randomly chosen 160-bit integer then, by Table 4.4, MILLER-RABIN($q, 18$) is a robust test for the primality of q . If p is assumed to be a randomly chosen L -bit integer, then by Table 4.4, MILLER-RABIN($p, 5$) is a robust test for the primality of p . Since the Miller-Rabin test is a probabilistic primality test, the output of Algorithm 4.56 is a probable prime.
- (ii) To improve performance, candidate primes q and p should be subjected to trial division by all odd primes less than some bound B before invoking the Miller-Rabin test. See Note 4.45 for guidance on selecting B .

4.58 Note (“weak” primes cannot be intentionally constructed) Algorithm 4.56 has the feature that the random seed s is not input to the prime number generation portion of the algorithm itself, but rather to an unpredictable and uncontrollable randomization process (steps 2.2 and 4.1), the output of which is used as the actual random seed. This precludes manipulation of the input seed to the prime number generation. If the seed s and counter i are made public, then anyone can verify that q and p were generated using the approved method. This feature prevents a central authority who generates p and q as system-wide parameters for use in the DSA from intentionally constructing “weak” primes q and p which it could subsequently exploit to recover other entities’ private keys.

4.4.4 Constructive techniques for provable primes

Maurer's algorithm (Algorithm 4.62) generates random *provable* primes that are almost uniformly distributed over the set of all primes of a specified size. The expected time for generating a prime is only slightly greater than that for generating a probable prime of equal size using Algorithm 4.44 with security parameter $t = 1$. (In practice, one may wish to choose $t > 1$ in Algorithm 4.44; cf. Note 4.49.)

The main idea behind Algorithm 4.62 is Fact 4.59, which is a slight modification of Pocklington's theorem (Fact 4.40) and Fact 4.41.

4.59 Fact Let $n \geq 3$ be an odd integer, and suppose that $n = 1 + 2Rq$ where q is an odd prime. Suppose further that $q > R$.

- (i) If there exists an integer a satisfying $a^{n-1} \equiv 1 \pmod{n}$ and $\gcd(a^{2R} - 1, n) = 1$, then n is prime.
- (ii) If n is prime, the probability that a randomly selected base a , $1 \leq a \leq n-1$, satisfies $a^{n-1} \equiv 1 \pmod{n}$ and $\gcd(a^{2R} - 1, n) = 1$ is $(1 - 1/q)$.

Algorithm 4.62 recursively generates an odd prime q , and then chooses random integers R , $R < q$, until $n = 2Rq + 1$ can be proven prime using Fact 4.59(i) for some base a . By Fact 4.59(ii) the proportion of such bases is $1 - 1/q$ for prime n . On the other hand, if n is composite, then most bases a will fail to satisfy the condition $a^{n-1} \equiv 1 \pmod{n}$.

4.60 Note (description of constants c and m in Algorithm 4.62)

- (i) The optimal value of the constant c defining the trial division bound $B = ck^2$ in step 2 depends on the implementation of long-integer arithmetic, and is best determined experimentally (cf. Note 4.45).
- (ii) The constant $m = 20$ ensures that I is at least 20 bits long and hence the interval from which R is selected, namely $[I + 1, 2I]$, is sufficiently large (for the values of k of practical interest) that it most likely contains at least one value R for which $n = 2Rq + 1$ is prime.

4.61 Note (relative size r of q with respect to n) The relative size r of q with respect to n is defined to be $r = \lg q / \lg n$. In order to assure that the generated prime n is chosen randomly with essentially uniform distribution from the set of all k -bit primes, the size of the prime factor q of $n - 1$ must be chosen according to the probability distribution of the largest prime factor of a randomly selected k -bit integer. Since q must be greater than R in order for Fact 4.59 to apply, the relative size r of q is restricted to being in the interval $[\frac{1}{2}, 1]$. It can be deduced from Fact 3.7(i) that the cumulative probability distribution of the relative size r of the largest prime factor of a large random integer, given that r is at least $\frac{1}{2}$, is $(1 + \lg r)$ for $\frac{1}{2} \leq r \leq 1$. In step 4 of Algorithm 4.62, the relative size r is generated according to this distribution by selecting a random number $s \in [0, 1]$ and then setting $r = 2^{s-1}$. If $k \leq 2m$ then r is chosen to be the smallest permissible value, namely $\frac{1}{2}$, in order to ensure that the interval from which R is selected is sufficiently large (cf. Note 4.60(ii)).

4.62 Algorithm Maurer's algorithm for generating provable primesPROVABLE_PRIME(k)INPUT: a positive integer k .OUTPUT: a k -bit prime number n .

1. (If k is small, then test random integers by trial division. A table of small primes may be precomputed for this purpose.)
If $k \leq 20$ then repeatedly do the following:
 - 1.1 Select a random k -bit odd integer n .
 - 1.2 Use trial division by all primes less than \sqrt{n} to determine whether n is prime.
 - 1.3 If n is prime then return(n).
2. Set $c \leftarrow 0.1$ and $m \leftarrow 20$ (see Note 4.60).
3. (Trial division bound) Set $B \leftarrow c \cdot k^2$ (see Note 4.60).
4. (Generate r , the size of q relative to n — see Note 4.61) If $k > 2m$ then repeatedly do the following: select a random number s in the interval $[0, 1]$, set $r \leftarrow 2^{s-1}$, until $(k - rk) > m$. Otherwise (i.e. $k \leq 2m$), set $r \leftarrow 0.5$.
5. Compute $q \leftarrow \text{PROVABLE_PRIME}(\lceil r \cdot k \rceil + 1)$.
6. Set $I \leftarrow \lfloor 2^{k-1} / (2q) \rfloor$.
7. success $\leftarrow 0$.
8. While (success = 0) do the following:
 - 8.1 (select a candidate integer n) Select a random integer R in the interval $[I + 1, 2I]$ and set $n \leftarrow 2Rq + 1$.
 - 8.2 Use trial division to determine whether n is divisible by any prime number $< B$. If it is not then do the following:
 - Select a random integer a in the interval $[2, n - 2]$.
 - Compute $b \leftarrow a^{n-1} \bmod n$.
 - If $b = 1$ then do the following:
 - Compute $b \leftarrow a^{2R} \bmod n$ and $d \leftarrow \gcd(b - 1, n)$.
 - If $d = 1$ then success $\leftarrow 1$.
9. Return(n).

4.63 Note (improvements to Algorithm 4.62)

- (i) A speedup can be achieved by using Fact 4.42 instead of Fact 4.59(i) for proving $n = 2Rq + 1$ prime in step 8.2 of Maurer's algorithm — Fact 4.42 only requires that q be greater than $\sqrt[3]{n}$.
- (ii) If a candidate n passes the trial division (in step 8.2), then a Miller-Rabin test (Algorithm 4.24) with the single base $a = 2$ should be performed on n ; only if n passes this test should the attempt to prove its primality (the remainder of step 8.2) be undertaken. This leads to a faster implementation due to the efficiency of the Miller-Rabin test with a single base $a = 2$ (cf. Remark 4.50).
- (iii) Step 4 requires the use of real number arithmetic when computing 2^{s-1} . To avoid these computations, one can precompute and store a list of such values for a selection of random numbers $s \in [0, 1]$.

4.64 Note (provable primes vs. probable primes) Probable primes are advantageous over provable primes in that Algorithm 4.44 for generating probable primes with $t = 1$ is slightly faster than Maurer's algorithm. Moreover, the latter requires more run-time memory due

to its recursive nature. Provable primes are preferable to probable primes in the sense that the former have zero error probability. In any cryptographic application, however, there is always a non-zero error probability of some catastrophic failure, such as the adversary guessing a secret key or hardware failure. Since the error probability of probable primes can be efficiently brought down to acceptably low levels (see Note 4.49 but note the dependence on t), there appears to be no reason for mandating the use of provable primes over probable primes.

4.5 Irreducible polynomials over \mathbb{Z}_p

Recall (Definition 2.190) that a polynomial $f(x) \in \mathbb{Z}_p[x]$ of degree $m \geq 1$ is said to be *irreducible over \mathbb{Z}_p* if it cannot be written as a product of two polynomials in $\mathbb{Z}_p[x]$ each having degree less than m . Such a polynomial $f(x)$ can be used to represent the elements of the finite field \mathbb{F}_{p^m} as $\mathbb{F}_{p^m} = \mathbb{Z}_p[x]/(f(x))$, the set of all polynomials in $\mathbb{Z}_p[x]$ of degree less than m where the addition and multiplication of polynomials is performed modulo $f(x)$ (see §2.6.3). This section presents techniques for constructing irreducible polynomials over \mathbb{Z}_p , where p is a prime. The characteristic two finite fields \mathbb{F}_{2^m} are of particular interest for cryptographic applications because the arithmetic in these fields can be efficiently performed both in software and in hardware. Thus, attention in the section will be focused on irreducible polynomials over \mathbb{Z}_2 .

The arithmetic in finite fields can usually be implemented more efficiently if the irreducible polynomial chosen has few non-zero terms. Irreducible *trinomials*, i.e., irreducible polynomials having exactly three non-zero terms, are considered in §4.5.2. *Primitive* polynomials, i.e., irreducible polynomials $f(x)$ of degree m in $\mathbb{Z}_p[x]$ for which x is a generator of $\mathbb{F}_{p^m}^*$, the multiplicative group of the finite field $\mathbb{F}_{p^m} = \mathbb{Z}_p[x]/(f(x))$ (Definition 2.228), are the topic of §4.5.3. Primitive polynomials are also used in the generation of linear feedback shift register sequences having the maximum possible period (Fact 6.12).

4.5.1 Irreducible polynomials

If $f(x) \in \mathbb{Z}_p[x]$ is irreducible over \mathbb{Z}_p and a is a non-zero element in \mathbb{Z}_p , then $a \cdot f(x)$ is also irreducible over \mathbb{Z}_p . Hence it suffices to restrict attention to *monic* polynomials in $\mathbb{Z}_p[x]$, i.e., polynomials whose leading coefficient is 1. Observe also that if $f(x)$ is an irreducible polynomial, then its constant term must be non-zero. In particular, if $f(x) \in \mathbb{Z}_2[x]$, then its constant term must be 1.

There is a formula for computing exactly the number of monic irreducible polynomials in $\mathbb{Z}_p[x]$ of a fixed degree. The Möbius function, which is defined next, is used in this formula.

4.65 Definition Let m be a positive integer. The *Möbius function* μ is defined by

$$\mu(m) = \begin{cases} 1, & \text{if } m = 1, \\ 0, & \text{if } m \text{ is divisible by the square of a prime,} \\ (-1)^k, & \text{if } m \text{ is the product of } k \text{ distinct primes.} \end{cases}$$

4.66 Example (Möbius function) The following table gives the values of the Möbius function $\mu(m)$ for the first 10 values of m :

m	1	2	3	4	5	6	7	8	9	10
$\mu(m)$	1	-1	-1	0	-1	1	-1	0	0	1

□

4.67 Fact (number of monic irreducible polynomials) Let p be a prime and m a positive integer.

- (i) The number $N_p(m)$ of monic irreducible polynomials of degree m in $\mathbb{Z}_p[x]$ is given by the following formula:

$$N_p(m) = \frac{1}{m} \sum_{d|m} \mu(d) p^{m/d},$$

where the summation ranges over all positive divisors d of m .

- (ii) The probability of a random monic polynomial of degree m in $\mathbb{Z}_p[x]$ being irreducible over \mathbb{Z}_p is roughly $\frac{1}{m}$. More specifically, the number $N_p(m)$ satisfies

$$\frac{1}{2m} \leq \frac{N_p(m)}{p^m} \approx \frac{1}{m}.$$

Testing irreducibility of polynomials in $\mathbb{Z}_p[x]$ is significantly simpler than testing primality of integers. A polynomial can be tested for irreducibility by verifying that it has no irreducible factors of degree $\leq \lfloor \frac{m}{2} \rfloor$. The following result leads to an efficient method (Algorithm 4.69) for accomplishing this.

4.68 Fact Let p be a prime and let k be a positive integer.

- (i) The product of all monic irreducible polynomials in $\mathbb{Z}_p[x]$ of degree dividing k is equal to $x^{p^k} - x$.
- (ii) Let $f(x)$ be a polynomial of degree m in $\mathbb{Z}_p[x]$. Then $f(x)$ is irreducible over \mathbb{Z}_p if and only if $\gcd(f(x), x^{p^i} - x) = 1$ for each i , $1 \leq i \leq \lfloor \frac{m}{2} \rfloor$.

4.69 Algorithm Testing a polynomial for irreducibility

INPUT: a prime p and a monic polynomial $f(x)$ of degree m in $\mathbb{Z}_p[x]$.

OUTPUT: an answer to the question: "Is $f(x)$ irreducible over \mathbb{Z}_p ?"

1. Set $u(x) \leftarrow x$.
 2. For i from 1 to $\lfloor \frac{m}{2} \rfloor$ do the following:
 - 2.1 Compute $u(x) \leftarrow u(x)^p \bmod f(x)$ using Algorithm 2.227. (Note that $u(x)$ is a polynomial in $\mathbb{Z}_p[x]$ of degree less than m .)
 - 2.2 Compute $d(x) = \gcd(f(x), u(x) - x)$ (using Algorithm 2.218).
 - 2.3 If $d(x) \neq 1$ then return("reducible").
 3. Return("irreducible").
-

Fact 4.67 suggests that one method for finding an irreducible polynomial of degree m in $\mathbb{Z}_p[x]$ is to generate a random monic polynomial of degree m in $\mathbb{Z}_p[x]$, test it for irreducibility, and continue until an irreducible one is found (Algorithm 4.70). The expected number of polynomials to be tried before an irreducible one is found is approximately m .

4.70 Algorithm Generating a random monic irreducible polynomial over \mathbb{Z}_p INPUT: a prime p and a positive integer m .OUTPUT: a monic irreducible polynomial $f(x)$ of degree m in $\mathbb{Z}_p[x]$.

1. Repeat the following:

1.1 (Generate a random monic polynomial of degree m in $\mathbb{Z}_p[x]$)Randomly select integers $a_0, a_1, a_2, \dots, a_{m-1}$ between 0 and $p-1$ with $a_0 \neq 0$. Let $f(x)$ be the polynomial $f(x) = x^m + a_{m-1}x^{m-1} + \dots + a_2x^2 + a_1x + a_0$.1.2 Use Algorithm 4.69 to test whether $f(x)$ is irreducible over \mathbb{Z}_p .Until $f(x)$ is irreducible.2. Return($f(x)$).

It is known that the expected degree of the irreducible factor of least degree of a random polynomial of degree m in $\mathbb{Z}_p[x]$ is $O(\lg m)$. Hence for each choice of $f(x)$, the expected number of times steps 2.1 – 2.3 of Algorithm 4.69 are iterated is $O(\lg m)$. Each iteration takes $O((\lg p)m^2)$ \mathbb{Z}_p -operations. These observations, together with Fact 4.67(ii), determine the running time for Algorithm 4.70.

4.71 Fact Algorithm 4.70 has an expected running time of $O(m^3(\lg m)(\lg p))$ \mathbb{Z}_p -operations.

Given one irreducible polynomial of degree m over \mathbb{Z}_p , Note 4.74 describes a method, which is more efficient than Algorithm 4.70, for randomly generating additional such polynomials,

4.72 Definition Let \mathbb{F}_q be a finite field of characteristic p , and let $\alpha \in \mathbb{F}_q$. A *minimum polynomial* of α over \mathbb{Z}_p is a monic polynomial of least degree in $\mathbb{Z}_p[x]$ having α as a root.**4.73 Fact** Let \mathbb{F}_q be a finite field of order $q = p^m$, and let $\alpha \in \mathbb{F}_q$.

- (i) The minimum polynomial of α over \mathbb{Z}_p , denoted $m_\alpha(x)$, is unique.
- (ii) $m_\alpha(x)$ is irreducible over \mathbb{Z}_p .
- (iii) The degree of $m_\alpha(x)$ is a divisor of m .
- (iv) Let t be the smallest positive integer such that $\alpha^{p^t} = \alpha$. (Note that such a t exists since, by Fact 2.213, $\alpha^{p^m} = \alpha$.) Then

$$m_\alpha(x) = \prod_{i=0}^{t-1} (x - \alpha^{p^i}). \quad (4.1)$$

4.74 Note (*generating new irreducible polynomials from a given one*) Suppose that $f(y)$ is a given irreducible polynomial of degree m over \mathbb{Z}_p . The finite field \mathbb{F}_{p^m} can then be represented as $\mathbb{F}_{p^m} = \mathbb{Z}_p[y]/(f(y))$. A random monic irreducible polynomial of degree m over \mathbb{Z}_p can be efficiently generated as follows. First generate a random element $\alpha \in \mathbb{F}_{p^m}$ and then, by repeated exponentiation by p , determine the smallest positive integer t for which $\alpha^{p^t} = \alpha$. If $t < m$, then generate a new random element $\alpha \in \mathbb{F}_{p^m}$ and repeat; the probability that $t < m$ is known to be at most $(\lg m)/q^{m/2}$. If indeed $t = m$, then compute $m_\alpha(x)$ using the formula (4.1). Then $m_\alpha(x)$ is a random monic irreducible polynomial of degree m in $\mathbb{Z}_p[x]$. This method has an expected running time of $O(m^3(\lg p))$ \mathbb{Z}_p -operations (compare with Fact 4.71).

4.5.2 Irreducible trinomials

If a polynomial $f(x)$ in $\mathbb{Z}_2[x]$ has an even number of non-zero terms, then $f(1) = 0$, whence $(x + 1)$ is a factor of $f(x)$. Hence, the smallest number of non-zero terms an irreducible polynomial of degree ≥ 2 in $\mathbb{Z}_2[x]$ can have is three. An irreducible *trinomial* of degree m in $\mathbb{Z}_2[x]$ must be of the form $x^m + x^k + 1$, where $1 \leq k \leq m - 1$. Choosing an irreducible trinomial $f(x) \in \mathbb{Z}_2[x]$ of degree m to represent the elements of the finite field $\mathbb{F}_{2^m} = \mathbb{Z}_2[x]/(f(x))$ can lead to a faster implementation of the field arithmetic. The following facts are sometimes of use when searching for irreducible trinomials.

4.75 Fact Let m be a positive integer, and let k denote an integer in the interval $[1, m - 1]$.

- (i) If the trinomial $x^m + x^k + 1$ is irreducible over \mathbb{Z}_2 then so is $x^m + x^{m-k} + 1$.
- (ii) If $m \equiv 0 \pmod{8}$, there is no irreducible trinomial of degree m in $\mathbb{Z}_2[x]$.
- (iii) Suppose that either $m \equiv 3 \pmod{8}$ or $m \equiv 5 \pmod{8}$. Then a necessary condition for $x^m + x^k + 1$ to be irreducible over \mathbb{Z}_2 is that either k or $m - k$ must be of the form $2d$ for some positive divisor d of m .

Tables 4.6 and 4.7 list an irreducible trinomial of degree m over \mathbb{Z}_2 for each $m \leq 1478$ for which such a trinomial exists.

4.5.3 Primitive polynomials

Primitive polynomials were introduced at the beginning of §4.5. Let $f(x) \in \mathbb{Z}_p[x]$ be an irreducible polynomial of degree m . If the factorization of the integer $p^m - 1$ is known, then Fact 4.76 yields an efficient algorithm (Algorithm 4.77) for testing whether or not $f(x)$ is a primitive polynomial. If the factorization of $p^m - 1$ is unknown, there is no efficient algorithm known for performing this test.

4.76 Fact Let p be a prime and let the distinct prime factors of $p^m - 1$ be r_1, r_2, \dots, r_t . Then an irreducible polynomial $f(x) \in \mathbb{Z}_p[x]$ is primitive if and only if for each i , $1 \leq i \leq t$:

$$x^{(p^m - 1)/r_i} \not\equiv 1 \pmod{f(x)}.$$

(That is, x is an element of order $p^m - 1$ in the field $\mathbb{Z}_p[x]/(f(x))$.)

4.77 Algorithm Testing whether an irreducible polynomial is primitive

INPUT: a prime p , a positive integer m , the distinct prime factors r_1, r_2, \dots, r_t of $p^m - 1$, and a monic irreducible polynomial $f(x)$ of degree m in $\mathbb{Z}_p[x]$.

OUTPUT: an answer to the question: "Is $f(x)$ a primitive polynomial?"

1. For i from 1 to t do the following:
 - 1.1 Compute $l(x) = x^{(p^m - 1)/r_i} \bmod f(x)$ (using Algorithm 2.227).
 - 1.2 If $l(x) = 1$ then return("not primitive").
2. Return("primitive").

There are precisely $\phi(p^m - 1)/m$ monic primitive polynomials of degree m in $\mathbb{Z}_p[x]$ (Fact 2.230), where ϕ is the Euler phi function (Definition 2.100). Since the number of monic irreducible polynomials of degree m in $\mathbb{Z}_p[x]$ is roughly p^m/m (Fact 4.67(ii)), it follows that the probability of a random monic irreducible polynomial of degree m in $\mathbb{Z}_p[x]$

m	k	m	k	m	k	m	k	m	k	m	k	m	k
2	1	93	2	193	15	295	48	402	171	508	9	618	295
3	1	94	21	194	87	297	5	404	65	510	69	620	9
4	1	95	11	196	3	300	5	406	141	511	10	622	297
5	2	97	6	198	9	302	41	407	71	513	26	623	68
6	1	98	11	199	34	303	1	409	87	514	67	625	133
7	1	100	15	201	14	305	102	412	147	516	21	626	251
9	1	102	29	202	55	308	15	414	13	518	33	628	223
10	3	103	9	204	27	310	93	415	102	519	79	631	307
11	2	105	4	207	43	313	79	417	107	521	32	633	101
12	3	106	15	209	6	314	15	418	199	522	39	634	39
14	5	108	17	210	7	316	63	420	7	524	167	636	217
15	1	110	33	212	105	318	45	422	149	526	97	639	16
17	3	111	10	214	73	319	36	423	25	527	47	641	11
18	3	113	9	215	23	321	31	425	12	529	42	642	119
20	3	118	33	217	45	322	67	426	63	532	1	646	249
21	2	119	8	218	11	324	51	428	105	534	161	647	5
22	1	121	18	220	7	327	34	431	120	537	94	649	37
23	5	123	2	223	33	329	50	433	33	538	195	650	3
25	3	124	19	225	32	330	99	436	165	540	9	651	14
28	1	126	21	228	113	332	89	438	65	543	16	652	93
29	2	127	1	231	26	333	2	439	49	545	122	654	33
30	1	129	5	233	74	337	55	441	7	550	193	655	88
31	3	130	3	234	31	340	45	444	81	551	135	657	38
33	10	132	17	236	5	342	125	446	105	553	39	658	55
34	7	134	57	238	73	343	75	447	73	556	153	660	11
35	2	135	11	239	36	345	22	449	134	558	73	662	21
36	9	137	21	241	70	346	63	450	47	559	34	663	107
39	4	140	15	242	95	348	103	455	38	561	71	665	33
41	3	142	21	244	111	350	53	457	16	564	163	668	147
42	7	145	52	247	82	351	34	458	203	566	153	670	153
44	5	146	71	249	35	353	69	460	19	567	28	671	15
46	1	147	14	250	103	354	99	462	73	569	77	673	28
47	5	148	27	252	15	358	57	463	93	570	67	676	31
49	9	150	53	253	46	359	68	465	31	574	13	679	66
52	3	151	3	255	52	362	63	468	27	575	146	682	171
54	9	153	1	257	12	364	9	470	9	577	25	684	209
55	7	154	15	258	71	366	29	471	1	580	237	686	197
57	4	155	62	260	15	367	21	473	200	582	85	687	13
58	19	156	9	263	93	369	91	474	191	583	130	689	14
60	1	159	31	265	42	370	139	476	9	585	88	690	79
62	29	161	18	266	47	372	111	478	121	588	35	692	299
63	1	162	27	268	25	375	16	479	104	590	93	694	169
65	18	166	37	270	53	377	41	481	138	593	86	695	177
66	3	167	6	271	58	378	43	484	105	594	19	697	267
68	9	169	34	273	23	380	47	486	81	596	273	698	215
71	6	170	11	274	67	382	81	487	94	599	30	700	75
73	25	172	1	276	63	383	90	489	83	601	201	702	37
74	35	174	13	278	5	385	6	490	219	602	215	705	17
76	21	175	6	279	5	386	83	492	7	604	105	708	15
79	9	177	8	281	93	388	159	494	17	606	165	711	92
81	4	178	31	282	35	390	9	495	76	607	105	713	41
84	5	180	3	284	53	391	28	497	78	609	31	714	23
86	21	182	81	286	69	393	7	498	155	610	127	716	183
87	13	183	56	287	71	394	135	500	27	612	81	718	165
89	38	185	24	289	21	396	25	503	3	614	45	719	150
90	27	186	11	292	37	399	26	505	156	615	211	721	9
92	21	191	9	294	33	401	152	506	23	617	200	722	231

Table 4.6: Irreducible trinomials $x^m + x^k + 1$ over \mathbb{Z}_2 . For each m , $1 \leq m \leq 722$, for which an irreducible trinomial of degree m in $\mathbb{Z}_2[x]$ exists, the table lists the smallest k for which $x^m + x^k + 1$ is irreducible over \mathbb{Z}_2 .

m	k	m	k	m	k	m	k	m	k	m	k	m	k
724	207	831	49	937	217	1050	159	1159	66	1265	119	1374	609
726	5	833	149	938	207	1052	291	1161	365	1266	7	1375	52
727	180	834	15	942	45	1054	105	1164	19	1268	345	1377	100
729	58	838	61	943	24	1055	24	1166	189	1270	333	1380	183
730	147	839	54	945	77	1057	198	1167	133	1271	17	1383	130
732	343	841	144	948	189	1058	27	1169	114	1273	168	1385	12
735	44	842	47	951	260	1060	439	1170	27	1276	217	1386	219
737	5	844	105	953	168	1062	49	1174	133	1278	189	1388	11
738	347	845	2	954	131	1063	168	1175	476	1279	216	1390	129
740	135	846	105	956	305	1065	463	1177	16	1281	229	1391	3
742	85	847	136	959	143	1071	7	1178	375	1282	231	1393	300
743	90	849	253	961	18	1078	361	1180	25	1284	223	1396	97
745	258	850	111	964	103	1079	230	1182	77	1286	153	1398	601
746	351	852	159	966	201	1081	24	1183	87	1287	470	1399	55
748	19	855	29	967	36	1082	407	1185	134	1289	99	1401	92
750	309	857	119	969	31	1084	189	1186	171	1294	201	1402	127
751	18	858	207	972	7	1085	62	1188	75	1295	38	1404	81
753	158	860	35	975	19	1086	189	1190	233	1297	198	1407	47
754	19	861	14	977	15	1087	112	1191	196	1298	399	1409	194
756	45	862	349	979	178	1089	91	1193	173	1300	75	1410	383
758	233	865	1	982	177	1090	79	1196	281	1302	77	1412	125
759	98	866	75	983	230	1092	23	1198	405	1305	326	1414	429
761	3	868	145	985	222	1094	57	1199	114	1306	39	1415	282
762	83	870	301	986	3	1095	139	1201	171	1308	495	1417	342
767	168	871	378	988	121	1097	14	1202	287	1310	333	1420	33
769	120	873	352	990	161	1098	83	1204	43	1311	476	1422	49
772	7	876	149	991	39	1100	35	1206	513	1313	164	1423	15
774	185	879	11	993	62	1102	117	1207	273	1314	19	1425	28
775	93	881	78	994	223	1103	65	1209	118	1319	129	1426	103
777	29	882	99	996	65	1105	21	1210	243	1321	52	1428	27
778	375	884	173	998	101	1106	195	1212	203	1324	337	1430	33
780	13	887	147	999	59	1108	327	1214	257	1326	397	1431	17
782	329	889	127	1001	17	1110	417	1215	302	1327	277	1433	387
783	68	890	183	1007	75	1111	13	1217	393	1329	73	1434	363
785	92	892	31	1009	55	1113	107	1218	91	1332	95	1436	83
791	30	894	173	1010	99	1116	59	1220	413	1334	617	1438	357
793	253	895	12	1012	115	1119	283	1223	255	1335	392	1441	322
794	143	897	113	1014	385	1121	62	1225	234	1337	75	1442	395
798	53	898	207	1015	186	1122	427	1226	167	1338	315	1444	595
799	25	900	1	1020	135	1126	105	1228	27	1340	125	1446	421
801	217	902	21	1022	317	1127	27	1230	433	1343	348	1447	195
804	75	903	35	1023	7	1129	103	1231	105	1345	553	1449	13
806	21	905	117	1025	294	1130	551	1233	151	1348	553	1452	315
807	7	906	123	1026	35	1134	129	1234	427	1350	237	1454	297
809	15	908	143	1028	119	1135	9	1236	49	1351	39	1455	52
810	159	911	204	1029	98	1137	277	1238	153	1353	371	1457	314
812	29	913	91	1030	93	1138	31	1239	4	1354	255	1458	243
814	21	916	183	1031	68	1140	141	1241	54	1356	131	1460	185
815	333	918	77	1033	108	1142	357	1242	203	1358	117	1463	575
817	52	919	36	1034	75	1145	227	1246	25	1359	98	1465	39
818	119	921	221	1036	411	1146	131	1247	14	1361	56	1466	311
820	123	924	31	1039	21	1148	23	1249	187	1362	655	1468	181
822	17	926	365	1041	412	1151	90	1252	97	1364	239	1470	49
823	9	927	403	1042	439	1153	241	1255	589	1366	1	1471	25
825	38	930	31	1044	41	1154	75	1257	289	1367	134	1473	77
826	255	932	177	1047	10	1156	307	1260	21	1369	88	1476	21
828	189	935	417	1049	141	1158	245	1263	77	1372	181	1478	69

Table 4.7: Irreducible trinomials $x^m + x^k + 1$ over \mathbb{Z}_2 . For each m , $723 \leq m \leq 1478$, for which an irreducible trinomial of degree m in $\mathbb{Z}_2[x]$ exists, the table gives the smallest k for which $x^m + x^k + 1$ is irreducible over \mathbb{Z}_2 .

being primitive is approximately $\phi(p^m - 1)/p^m$. Using the lower bound for the Euler phi function (Fact 2.102), this probability can be seen to be at least $1/(6 \ln \ln p^m)$. This suggests the following algorithm for generating primitive polynomials.

4.78 Algorithm Generating a random monic primitive polynomial over \mathbb{Z}_p

INPUT: a prime p , integer $m \geq 1$, and the distinct prime factors r_1, r_2, \dots, r_t of $p^m - 1$.

OUTPUT: a monic primitive polynomial $f(x)$ of degree m in $\mathbb{Z}_p[x]$.

1. Repeat the following:
 - 1.1 Use Algorithm 4.70 to generate a random monic irreducible polynomial $f(x)$ of degree m in $\mathbb{Z}_p[x]$.
 - 1.2 Use Algorithm 4.77 to test whether $f(x)$ is primitive.
 Until $f(x)$ is primitive.
 2. Return($f(x)$).
-

For each m , $1 \leq m \leq 229$, Table 4.8 lists a polynomial of degree m that is primitive over \mathbb{Z}_2 . If there exists a primitive trinomial $f(x) = x^m + x^k + 1$, then the trinomial with the smallest k is listed. If no primitive trinomial exists, then a primitive pentanomial of the form $f(x) = x^m + x^{k_1} + x^{k_2} + x^{k_3} + 1$ is listed.

If $p^m - 1$ is prime, then Fact 4.76 implies that every irreducible polynomial of degree m in $\mathbb{Z}_p[x]$ is also primitive. Table 4.9 gives either a primitive trinomial or a primitive pentanomial of degree m over \mathbb{Z}_2 where m is an exponent of one of the first 27 Mersenne primes (Definition 4.35).

4.6 Generators and elements of high order

Recall (Definition 2.169) that if G is a (multiplicative) finite group, the *order* of an element $a \in G$ is the least positive integer t such that $a^t = 1$. If there are n elements in G , and if $a \in G$ is an element of order n , then G is said to be *cyclic* and a is called a *generator* or a *primitive element* of G (Definition 2.167). Of special interest for cryptographic applications are the multiplicative group \mathbb{Z}_p^* of the integers modulo a prime p , and the multiplicative group $\mathbb{F}_{2^m}^*$ of the finite field \mathbb{F}_{2^m} of characteristic two; these groups are cyclic (Fact 2.213). Also of interest is the group \mathbb{Z}_n^* (Definition 2.124), where n is the product of two distinct odd primes. This section deals with the problem of finding generators and other elements of high order in \mathbb{Z}_p^* , $\mathbb{F}_{2^m}^*$, and \mathbb{Z}_n^* . See §2.5.1 for background in group theory and §2.6 for background in finite fields.

Algorithm 4.79 is an efficient method for determining the order of a group element, given the prime factorization of the group order n . The correctness of the algorithm follows from the fact that the order of an element must divide n (Fact 2.171).

m	k or (k_1, k_2, k_3)	m	k or (k_1, k_2, k_3)	m	k or (k_1, k_2, k_3)	m	k or (k_1, k_2, k_3)
2	1	59	22, 21, 1	116	71, 70, 1	173	100, 99, 1
3	1	60	1	117	20, 18, 2	174	13
4	1	61	16, 15, 1	118	33	175	6
5	2	62	57, 56, 1	119	8	176	119, 118, 1
6	1	63	1	120	118, 111, 7	177	8
7	1	64	4, 3, 1	121	18	178	87
8	6, 5, 1	65	18	122	60, 59, 1	179	34, 33, 1
9	4	66	10, 9, 1	123	2	180	37, 36, 1
10	3	67	10, 9, 1	124	37	181	7, 6, 1
11	2	68	9	125	108, 107, 1	182	128, 127, 1
12	7, 4, 3	69	29, 27, 2	126	37, 36, 1	183	56
13	4, 3, 1	70	16, 15, 1	127	1	184	102, 101, 1
14	12, 11, 1	71	6	128	29, 27, 2	185	24
15	1	72	53, 47, 6	129	5	186	23, 22, 1
16	5, 3, 2	73	25	130	3	187	58, 57, 1
17	3	74	16, 15, 1	131	48, 47, 1	188	74, 73, 1
18	7	75	11, 10, 1	132	29	189	127, 126, 1
19	6, 5, 1	76	36, 35, 1	133	52, 51, 1	190	18, 17, 1
20	3	77	31, 30, 1	134	57	191	9
21	2	78	20, 19, 1	135	11	192	28, 27, 1
22	1	79	9	136	126, 125, 1	193	15
23	5	80	38, 37, 1	137	21	194	87
24	4, 3, 1	81	4	138	8, 7, 1	195	10, 9, 1
25	3	82	38, 35, 3	139	8, 5, 3	196	66, 65, 1
26	8, 7, 1	83	46, 45, 1	140	29	197	62, 61, 1
27	8, 7, 1	84	13	141	32, 31, 1	198	65
28	3	85	28, 27, 1	142	21	199	34
29	2	86	13, 12, 1	143	21, 20, 1	200	42, 41, 1
30	16, 15, 1	87	13	144	70, 69, 1	201	14
31	3	88	72, 71, 1	145	52	202	55
32	28, 27, 1	89	38	146	60, 59, 1	203	8, 7, 1
33	13	90	19, 18, 1	147	38, 37, 1	204	74, 73, 1
34	15, 14, 1	91	84, 83, 1	148	27	205	30, 29, 1
35	2	92	13, 12, 1	149	110, 109, 1	206	29, 28, 1
36	11	93	2	150	53	207	43
37	12, 10, 2	94	21	151	3	208	62, 59, 3
38	6, 5, 1	95	11	152	66, 65, 1	209	6
39	4	96	49, 47, 2	153	1	210	35, 32, 3
40	21, 19, 2	97	6	154	129, 127, 2	211	46, 45, 1
41	3	98	11	155	32, 31, 1	212	105
42	23, 22, 1	99	47, 45, 2	156	116, 115, 1	213	8, 7, 1
43	6, 5, 1	100	37	157	27, 26, 1	214	49, 48, 1
44	27, 26, 1	101	7, 6, 1	158	27, 26, 1	215	23
45	4, 3, 1	102	77, 76, 1	159	31	216	196, 195, 1
46	21, 20, 1	103	9	160	19, 18, 1	217	45
47	5	104	11, 10, 1	161	18	218	11
48	28, 27, 1	105	16	162	88, 87, 1	219	19, 18, 1
49	9	106	15	163	60, 59, 1	220	15, 14, 1
50	27, 26, 1	107	65, 63, 2	164	14, 13, 1	221	35, 34, 1
51	16, 15, 1	108	31	165	31, 30, 1	222	92, 91, 1
52	3	109	7, 6, 1	166	39, 38, 1	223	33
53	16, 15, 1	110	13, 12, 1	167	6	224	31, 30, 1
54	37, 36, 1	111	10	168	17, 15, 2	225	32
55	24	112	45, 43, 2	169	34	226	58, 57, 1
56	22, 21, 1	113	9	170	23	227	46, 45, 1
57	7	114	82, 81, 1	171	19, 18, 1	228	148, 147, 1
58	19	115	15, 14, 1	172	7	229	64, 63, 1

Table 4.8: Primitive polynomials over \mathbb{Z}_2 . For each m , $1 \leq m \leq 229$, an exponent k is given for which the trinomial $x^m + x^k + 1$ is primitive over \mathbb{Z}_2 . If no such trinomial exists, a triple of exponents (k_1, k_2, k_3) is given for which the pentanomial $x^m + x^{k_1} + x^{k_2} + x^{k_3} + 1$ is primitive over \mathbb{Z}_2 .

j	m	$k(k_1, k_2, k_3)$
1	2	1
2	3	1
3	5	2
4	7	1, 3
5	13	none (4,3,1)
6	17	3, 5, 6
7	19	none (5,2,1)
8	31	3, 6, 7, 13
9	61	none (43,26,14)
10	89	38
11	107	none (82,57,31)
12	127	1, 7, 15, 30, 63
13	521	32, 48, 158, 168
14	607	105, 147, 273
15	1279	216, 418
16	2203	none (1656,1197,585)
17	2281	715, 915, 1029
18	3217	67, 576
19	4253	none (3297,2254,1093)
20	4423	271, 369, 370, 649, 1393, 1419, 2098
21	9689	84, 471, 1836, 2444, 4187
22	9941	none (7449,4964,2475)
23	11213	none (8218,6181,2304)
24	19937	881, 7083, 9842
25	21701	none (15986,11393,5073)
26	23209	1530, 6619, 9739
27	44497	8575, 21034

Table 4.9: Primitive polynomials of degree m over \mathbb{Z}_2 , $2^m - 1$ a Mersenne prime. For each exponent $m = M_j$ of the first 27 Mersenne primes, the table lists all values of k , $1 \leq k \leq m/2$, for which the trinomial $x^m + x^k + 1$ is irreducible over \mathbb{Z}_2 . If no such trinomial exists, a triple of exponents (k_1, k_2, k_3) is listed such that the pentanomial $x^m + x^{k_1} + x^{k_2} + x^{k_3} + 1$ is irreducible over \mathbb{Z}_2 .

4.79 Algorithm Determining the order of a group element

INPUT: a (multiplicative) finite group G of order n , an element $a \in G$, and the prime factorization $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$.

OUTPUT: the order t of a .

1. Set $t \leftarrow n$.
2. For i from 1 to k do the following:
 - 2.1 Set $t \leftarrow t/p_i^{e_i}$.
 - 2.2 Compute $a_1 \leftarrow a^t$.
 - 2.3 While $a_1 \neq 1$ do the following: compute $a_1 \leftarrow a_1^{p_i}$ and set $t \leftarrow t \cdot p_i$.
3. Return(t).

Suppose now that G is a cyclic group of order n . Then for any divisor d of n the number of elements of order d in G is exactly $\phi(d)$ (Fact 2.173(ii)), where ϕ is the Euler phi function (Definition 2.100). In particular, G has exactly $\phi(n)$ generators, and hence the probability of a random element in G being a generator is $\phi(n)/n$. Using the lower bound for the Euler phi function (Fact 2.102), this probability can be seen to be at least $1/(6 \ln \ln n)$. This

suggests the following efficient randomized algorithm for finding a generator of a cyclic group.

4.80 Algorithm Finding a generator of a cyclic group

INPUT: a cyclic group G of order n , and the prime factorization $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$.

OUTPUT: a generator α of G .

1. Choose a random element α in G .
 2. For i from 1 to k do the following:
 - 2.1 Compute $b \leftarrow \alpha^{n/p_i}$.
 - 2.2 If $b = 1$ then go to step 1.
 3. Return(α).
-

4.81 Note (*group elements of high order*) In some situations it may be desirable to have an element of high order, and not a generator. Given a generator α in a cyclic group G of order n , and given a divisor d of n , an element β of order d in G can be efficiently obtained as follows: $\beta = \alpha^{n/d}$. If q is a prime divisor of the order n of a cyclic group G , then the following method finds an element $\beta \in G$ of order q without first having to find a generator of G : select a random element $g \in G$ and compute $\beta = g^{n/q}$; repeat until $\beta \neq 1$.

4.82 Note (*generators of $\mathbb{F}_{2^m}^*$*) There are two basic approaches to finding a generator of $\mathbb{F}_{2^m}^*$. Both techniques require the factorization of the order of $\mathbb{F}_{2^m}^*$, namely $2^m - 1$.

- (i) Generate a monic primitive polynomial $f(x)$ of degree m over \mathbb{Z}_2 (Algorithm 4.78). The finite field \mathbb{F}_{2^m} can then be represented as $\mathbb{Z}_2[x]/(f(x))$, the set of all polynomials over \mathbb{Z}_2 modulo $f(x)$, and the element $\alpha = x$ is a generator.
- (ii) Select the method for representing elements of \mathbb{F}_{2^m} first. Then use Algorithm 4.80 with $G = \mathbb{F}_{2^m}^*$ and $n = 2^m - 1$ to find a generator α of $\mathbb{F}_{2^m}^*$.

If $n = pq$, where p and q are distinct odd primes, then \mathbb{Z}_n^* is a non-cyclic group of order $\phi(n) = (p-1)(q-1)$. The maximum order of an element in \mathbb{Z}_n^* is $\text{lcm}(p-1, q-1)$. Algorithm 4.83 is a method for generating such an element which requires the factorizations of $p-1$ and $q-1$.

4.83 Algorithm Selecting an element of maximum order in \mathbb{Z}_n^* where $n = pq$

INPUT: two distinct odd primes, p, q , and the factorizations of $p-1$ and $q-1$.

OUTPUT: an element α of maximum order $\text{lcm}(p-1, q-1)$ in \mathbb{Z}_n^* , where $n = pq$.

1. Use Algorithm 4.80 with $G = \mathbb{Z}_p^*$ and $n = p-1$ to find a generator a of \mathbb{Z}_p^* .
 2. Use Algorithm 4.80 with $G = \mathbb{Z}_q^*$ and $n = q-1$ to find a generator b of \mathbb{Z}_q^* .
 3. Use Gauss's algorithm (Algorithm 2.121) to find an integer α , $1 \leq \alpha \leq n-1$, satisfying $\alpha \equiv a \pmod{p}$ and $\alpha \equiv b \pmod{q}$.
 4. Return(α).
-

4.6.1 Selecting a prime p and generator of \mathbb{Z}_p^*

In cryptographic applications for which a generator of \mathbb{Z}_p^* is required, one usually has the flexibility of selecting the prime p . To guard against the Pohlig-Hellman algorithm for computing discrete logarithms (Algorithm 3.63), a security requirement is that $p-1$ should contain a "large" prime factor q . In this context, "large" means that the quantity \sqrt{q} represents an infeasible amount of computation; for example, $q \geq 2^{160}$. This suggests the following algorithm for selecting appropriate parameters (p, α) .

4.84 Algorithm Selecting a k -bit prime p and a generator α of \mathbb{Z}_p^*

INPUT: the required bitlength k of the prime and a security parameter t . \

OUTPUT: a k -bit prime p such that $p-1$ has a prime factor $\geq t$, and a generator α of \mathbb{Z}_p^* .

1. Repeat the following:
 - 1.1 Select a random k -bit prime p (for example, using Algorithm 4.44).
 - 1.2 Factor $p-1$.

Until $p-1$ has a prime factor $\geq t$.
 2. Use Algorithm 4.80 with $G = \mathbb{Z}_p^*$ and $n = p-1$ to find a generator α of \mathbb{Z}_p^* .
 3. Return(p, α).
-

Algorithm 4.84 is relatively inefficient as it requires the use of an integer factorization algorithm in step 1.2. An alternative approach is to generate the prime p by first choosing a large prime q and then selecting relatively small integers R at random until $p = 2Rq + 1$ is prime. Since $p-1 = 2Rq$, the factorization of $p-1$ can be obtained by factoring R . A particularly convenient situation occurs by imposing the condition $R = 1$. In this case the factorization of $p-1$ is simply $2q$. Furthermore, since $\phi(p-1) = \phi(2q) = \phi(2)\phi(q) = q-1$, the probability that a randomly selected element $\alpha \in \mathbb{Z}_p^*$ is a generator is $\frac{q-1}{2q} \approx \frac{1}{2}$.

4.85 Definition A *safe prime* p is a prime of the form $p = 2q + 1$ where q is prime.

Algorithm 4.86 generates a safe (probable) prime p and a generator of \mathbb{Z}_p^* .

4.86 Algorithm Selecting a k -bit safe prime p and a generator α of \mathbb{Z}_p^*

INPUT: the required bitlength k of the prime.

OUTPUT: a k -bit safe prime p and a generator α of \mathbb{Z}_p^* .

1. Do the following:
 - 1.1 Select a random $(k-1)$ -bit prime q (for example, using Algorithm 4.44).
 - 1.2 Compute $p \leftarrow 2q + 1$, and test whether p is prime (for example, using trial division by small primes and Algorithm 4.24).

Until p is prime.
 2. Use Algorithm 4.80 to find a generator α of \mathbb{Z}_p^* .
 3. Return(p, α).
-

4.7 Notes and further references

§4.1

Several books provide extensive treatments of primality testing including those by Bresoud [198], Bach and Shallit [70], and Koblitz [697]. The book by Kranakis [710] offers a more theoretical approach. Cohen [263] gives a comprehensive treatment of modern primality tests. See also the survey articles by A. Lenstra [747] and A. Lenstra and H. Lenstra [748]. Facts 4.1 and 4.2 were proven in 1837 by Dirichlet. For proofs of these results, see Chapter 16 of Ireland and Rosen [572]. Fact 4.3 is due to Rosser and Schoenfeld [1070]. Bach and Shallit [70] have further results on the distribution of prime numbers.

§4.2

Fact 4.13(i) was proven by Alford, Granville, and Pomerance [24]; see also Granville [521]. Fact 4.13(ii) is due to Pomerance, Selfridge, and Wagstaff [996]. Pinch [974] showed that there are 105212 Carmichael numbers up to 10^{15} .

The Solovay-Strassen probabilistic primality test (Algorithm 4.18) is due to Solovay and Strassen [1163], as modified by Atkin and Larson [57].

Fact 4.23 was proven independently by Monier [892] and Rabin [1024]. The Miller-Rabin test (Algorithm 4.24) originated in the work of Miller [876] who presented it as a non-probabilistic polynomial-time algorithm assuming the correctness of the Extended Riemann Hypothesis (ERH). Rabin [1021, 1024] rephrased Miller's algorithm as a probabilistic primality test. Rabin's algorithm required a small number of gcd computations. The Miller-Rabin test (Algorithm 4.24) is a simplification of Rabin's algorithm which does not require any gcd computations, and is due to Knuth [692, p.379]. Arazi [55], making use of Montgomery modular multiplication (§14.3.2), showed how the Miller-Rabin test can be implemented by "divisionless modular exponentiations" only, yielding a probabilistic primality test which does not use any division operations.

Miller [876], appealing to the work of Ankeny [32], proved under assumption of the Extended Riemann Hypothesis that, if n is an odd composite integer, then its least strong witness is less than $c(\ln n)^2$, where c is some constant. Bach [63] proved that this constant may be taken to be $c = 2$; see also Bach [64]. As a consequence, one can test n for primality in $O((\lg n)^5)$ bit operations by executing the Miller-Rabin algorithm for all bases $a \leq 2(\ln n)^2$. This gives a deterministic polynomial-time algorithm for primality testing, under the assumption that the ERH is true.

Table 4.1 is from Jaeschke [630], building on earlier work of Pomerance, Selfridge, and Wagstaff [996]. Arnault [56] found the following 46-digit composite integer

$$n = 1195068768795265792518361315725116351898245581$$

that is a strong pseudoprime to all the 11 prime bases up to 31. Arnault also found a 337-digit composite integer which is a strong pseudoprime to all 46 prime bases up to 199.

The Miller-Rabin test (Algorithm 4.24) randomly generates t independent bases a and tests to see if each is a strong witness for n . Let n be an odd composite integer and let $t = \lceil \frac{1}{2} \lg n \rceil$. In situations where random bits are scarce, one may choose instead to generate a single random base a and use the bases $a, a+1, \dots, a+t-1$. Bach [66] proved that for a randomly chosen integer a , the probability that $a, a+1, \dots, a+t-1$ are all strong liars for n is bounded above by $n^{-1/4+o(1)}$; in other words, the probability that the Miller-Rabin algorithm using these bases mistakenly declares an odd composite integer "prime" is at most $n^{-1/4+o(1)}$. Peralta and Shoup [969] later improved this bound to $n^{-1/2+o(1)}$.

Monier [892] gave exact formulas for the number of Fermat liars, Euler liars, and strong liars for composite integers. One consequence of Monier's formulas is the following improvement (in the case where n is not a prime power) of Fact 4.17 (see Kranakis [710, p.68]). If $n \geq 3$ is an odd composite integer having r distinct prime factors, and if $n \equiv 3 \pmod{4}$, then there are at most $\phi(n)/2^{r-1}$ Euler liars for n . Another consequence is the following improvement (in the case where n has at least three distinct prime factors) of Fact 4.23. If $n \geq 3$ is an odd composite integer having r distinct prime factors, then there are at most $\phi(n)/2^{r-1}$ strong liars for n . Erdős and Pomerance [373] estimated the average number of Fermat liars, Euler liars, and strong liars for composite integers. Fact 4.30(ii) was proven independently by Atkin and Larson [57], Monier [892], and Pomerance, Selfridge, and Wagstaff [996].

Pinch [975] reviewed the probabilistic primality tests used in the *Mathematica*, *Maple V*, *Axiom*, and *Pari/GP* computer algebra systems. Some of these systems use a probabilistic primality test known as the *Lucas test*; a description of this test is provided by Pomerance, Selfridge, and Wagstaff [996].

§4.3

If a number n is composite, providing a non-trivial divisor of n is evidence of its compositeness that can be verified in polynomial time (by long division). In other words, the decision problem "is n composite?" belongs to the complexity class NP (cf. Example 2.65). Pratt [1000] used Fact 4.38 to show that this decision problem is also in co-NP. That is, if n is prime there exists some evidence of this (called a *certificate of primality*) that can be verified in polynomial time. Note that the issue here is not in finding such evidence, but rather in determining whether such evidence exists which, if found, allows efficient verification. Pomerance [992] improved Pratt's results and showed that every prime n has a certificate of primality which requires $O(\ln n)$ multiplications modulo n for its verification.

Primality of the *Fermat number* $F_k = 2^{2^k} + 1$ can be determined in deterministic polynomial time by *Pepin's test*: for $k \geq 2$, F_k is prime if and only if $5^{(F_k-1)/2} \equiv -1 \pmod{F_k}$. For the history behind Pepin's test and the Lucas-Lehmer test (Algorithm 4.37), see Bach and Shallit [70].

In Fact 4.38, the integer a does not have to be the same for all q . More precisely, Brillhart and Selfridge [212] showed that Fact 4.38 can be refined as follows: an integer $n \geq 3$ is prime if and only if for each prime divisor q of $n-1$, there exists an integer a_q such that $a_q^{n-1} \equiv 1 \pmod{n}$ and $a_q^{(n-1)/q} \not\equiv 1 \pmod{n}$. The same is true of Fact 4.40, which is due to Pocklington [981]. For a proof of Fact 4.41, see Maurer [818]. Fact 4.42 is due to Brillhart, Lehmer, and Selfridge [210]; a simplified proof is given by Maurer [818].

The original Jacobi sum test was discovered by Adleman, Pomerance, and Rumely [16]. The algorithm was simplified, both theoretically and algorithmically, by Cohen and H. Lenstra [265]. Cohen and A. Lenstra [264] give an implementation report of the Cohen-Lenstra Jacobi sum test; see also Chapter 9 of Cohen [263]. Further improvements of the Jacobi sum test are reported by Bosma and van der Hulst [174].

Elliptic curves were first used for primality proving by Goldwasser and Kilian [477], who presented a randomized algorithm which has an expected running time of $O((\ln n)^8)$ bit operations for *most* inputs n . Subsequently, Adleman and Huang [13] designed a primality proving algorithm using hyperelliptic curves of genus two whose expected running time is polynomial for *all* inputs n . This established that the decision problem "is n prime?" is in the complexity class RP (Definition 2.77(ii)). The Goldwasser-Kilian and Adleman-Huang algorithms are inefficient in practice. Atkin's test, and an implementation of it, is extensively described by Atkin and Morain [58]; see also Chapter 9 of Cohen [263]. The

largest number proven prime as of 1996 by a general purpose primality proving algorithm is a 1505-decimal digit number, accomplished by Morain [903] using Atkin's test. The total time for the computation was estimated to be 4 years of CPU time distributed among 21 SUN 3/60 workstations. See also Morain [902] for an implementation report on Atkin's test which was used to prove the primality of the 1065-decimal digit number $(2^{3539} + 1)/3$.

§4.4

A proof of Merten's theorem can be found in Hardy and Wright [540]. The optimal trial division bound (Note 4.45) was derived by Maurer [818]. The discussion (Note 4.47) on the probability $P(X|Y_t)$ is from Beauchemin et al. [81]; the result mentioned in the last sentence of this note is due to Kim and Pomerance [673]. Fact 4.48 was derived by Damgård, Landrock, and Pomerance [300], building on earlier work of Erdős and Pomerance [373], Kim and Pomerance [673], and Damgård and Landrock [299]. Table 4.3 is Table 2 of Damgård, Landrock, and Pomerance [300]. The suggestions to first do a Miller-Rabin test with base $a = 2$ (Remark 4.50) and to do an incremental search (Note 4.51) in Algorithm 4.44 were made by Brandt, Damgård, and Landrock [187]. The error and failure probabilities for incremental search (Note 4.51(i)) were obtained by Brandt and Damgård [186]; consult this paper for more concrete estimates of these probabilities.

Algorithm 4.53 for generating strong primes is due to Gordon [514, 513]. Gordon originally proposed computing $p_0 = (s^{r-1} - r^{s-1}) \bmod rs$ in step 3. Kaliski (personal communication, April 1996) proposed the modified formula $p_0 = (2s^{r-2} \bmod r)s - 1$ which can be computed more efficiently. Williams and Schmid [1249] proposed an algorithm for generating strong primes p with the additional constraint that $p - 1 = 2q$ where q is prime; this algorithm is not as efficient as Gordon's algorithm. Hellman and Bach [550] recommended an additional constraint on strong primes, specifying that $s - 1$ (where s is a large prime factor of $p + 1$) must have a large prime factor (see §15.2.3(v)); this thwarts cycling attacks based on Lucas sequences.

The NIST method for prime generation (Algorithm 4.56) is that recommended by the NIST Federal Information Processing Standards Publication (FIPS) 186 [406].

Fact 4.59 and Algorithm 4.62 for provable prime generation are derived from Maurer [818]. Algorithm 4.62 is based on that of Shawe-Taylor [1123]. Maurer notes that the total diversity of reachable primes using the original version of his algorithm is roughly 10% of all primes. Maurer also presents a more complicated algorithm for generating provable primes with a better diversity than Algorithm 4.62, and provides extensive implementation details and analysis of the expected running time. Maurer [812] provides heuristic justification that Algorithm 4.62 generates primes with virtually uniform distribution. Mihailescu [870] observed that Maurer's algorithm can be improved by using the Eratosthenes sieve method for trial division (in step 8.2 of Algorithm 4.62) and by searching for a prime n in an appropriate interval of the arithmetic progression $2q + 1, 4q + 1, 6q + 1, \dots$ instead of generating R 's at random until $n = 2Rq + 1$ is prime. The second improvement comes at the expense of a reduction of the set of primes which may be produced by the algorithm. Mihailescu's paper includes extensive analysis and an implementation report.

§4.5

Lidl and Niederreiter [764] provide a comprehensive treatment of irreducible polynomials; proofs of Facts 4.67 and 4.68 can be found there.

Algorithm 4.69 for testing a polynomial for irreducibility is due to Ben-Or [109]. The fastest algorithm known for generating irreducible polynomials is due to Shoup [1131] and has an expected running time of $O(m^3 \lg m + m^2 \lg p)$ \mathbb{Z}_p -operations. There is no *deterministic* polynomial-time algorithm known for finding an irreducible polynomial of a specified

degree m in $\mathbb{Z}_p[x]$. Adleman and Lenstra [14] give a deterministic algorithm that runs in polynomial time under the assumption that the ERH is true. The best deterministic algorithm known is due to Shoup [1129] and takes $O(m^4 \sqrt{p})$ \mathbb{Z}_p -operations, ignoring powers of $\log m$ and $\log p$. Gordon [512] presents an improved method for computing minimum polynomials of elements in \mathbb{F}_{2^m} .

Zierler and Brillhart [1271] provide a table of all irreducible trinomials of degree ≤ 1000 in $\mathbb{Z}_2[x]$. Blake, Gao, and Lambert [146] extended this list to all irreducible trinomials of degree ≤ 2000 in $\mathbb{Z}_2[x]$. Fact 4.75 is from their paper.

Table 4.8 extends a similar table by Stahnke [1168]. The primitive pentanomials $x^m + x^{k_1} + x^{k_2} + x^{k_3} + 1$ listed in Table 4.8 have the following properties: (i) $k_1 = k_2 + k_3$; (ii) $k_2 > k_3$; and (iii) k_3 is as small as possible, and for this particular value of k_3 , k_2 is as small as possible. The rationale behind this form is explained in Stahnke's paper. For each $m < 5000$ for which the factorization of $2^m - 1$ is known, Živković [1275, 1276] gives a primitive trinomial in $\mathbb{Z}_2[x]$, one primitive polynomial in $\mathbb{Z}_2[x]$ having five non-zero terms, and one primitive polynomial in $\mathbb{Z}_2[x]$ having seven non-zero terms, provided that such polynomials exist. The factorizations of $2^m - 1$ are known for all $m \leq 510$ and for some additional $m \leq 5000$. A list of such factorizations can be found in Brillhart et al. [211] and updates of the list are available by anonymous ftp from `sable.ox.ac.uk` in the `/pub/math/cunningham/` directory. Hansen and Mullen [538] describe some improvements to Algorithm 4.78 for generating primitive polynomials. They also give tables of primitive polynomials of degree m in $\mathbb{Z}_p[x]$ for each prime power $p^m \leq 10^{50}$ with $p \leq 97$. Moreover, for each such p and m , the primitive polynomial of degree m over \mathbb{Z}_p listed has the smallest number of non-zero coefficients among all such polynomials.

The entries of Table 4.9 were obtained from Zierler [1270] for Mersenne exponents M_j , $1 \leq j \leq 23$, and from Kurita and Matsumoto [719] for Mersenne exponents M_j , $24 \leq j \leq 27$.

Let $f(x) \in \mathbb{Z}_p[x]$ be an irreducible polynomial of degree m , and consider the finite field $\mathbb{F}_{p^m} = \mathbb{Z}_p[x]/(f(x))$. Then $f(x)$ is called a *normal polynomial* if the set $\{x, x^p, x^{p^2}, \dots, x^{p^{m-1}}\}$ forms a basis for \mathbb{F}_{p^m} over \mathbb{Z}_p ; such a basis is called a *normal basis*. Mullin et al. [911] introduced the concept of an *optimal normal basis* in order to reduce the hardware complexity of multiplying field elements in the finite field \mathbb{F}_{2^m} . A VLSI implementation of the arithmetic in \mathbb{F}_{2^m} which uses optimal normal bases is described by Agnew et al. [18]. A normal polynomial which is also primitive is called a *primitive normal polynomial*. Davenport [301] proved that for any prime p and positive integer m there exists a primitive normal polynomial of degree m in $\mathbb{Z}_p[x]$. See also Lenstra and Schoof [760] who generalized this result from prime fields \mathbb{Z}_p to prime power fields \mathbb{F}_q . Morgan and Mullen [905] give a primitive normal polynomial of degree m over \mathbb{Z}_p for each prime power $p^m \leq 10^{50}$ with $p \leq 97$. Moreover, each polynomial has the smallest number of non-zero coefficients among all primitive normal polynomials of degree m over \mathbb{Z}_p ; in fact, each polynomial has at most five non-zero terms.

§4.6

No polynomial-time algorithm is known for finding generators, or even for testing whether an element is a generator, of a finite field \mathbb{F}_q if the factorization of $q - 1$ is unknown. Shoup [1130] considered the problem of deterministically generating in polynomial time a subset of \mathbb{F}_q that contains a generator, and presented a solution to the problem for the case where the characteristic p of \mathbb{F}_q is small (e.g. $p = 2$). Maurer [818] discusses how his algorithm (Algorithm 4.62) can be used to generate the parameters (p, α) , where p is a *provable* prime and α is a generator of \mathbb{Z}_p^* .

Chapter 5

Pseudorandom Bits and Sequences

Contents in Brief

5.1	Introduction	169
5.2	Random bit generation	171
5.3	Pseudorandom bit generation	173
5.4	Statistical tests	175
5.5	Cryptographically secure pseudorandom bit generation	185
5.6	Notes and further references	187

5.1 Introduction

The security of many cryptographic systems depends upon the generation of unpredictable quantities. Examples include the keystream in the one-time pad (§1.5.4), the secret key in the DES encryption algorithm (§11.5.1), the primes p, q in the RSA encryption (§8.2) and digital signature (§11.3.1) schemes, the private key a in the DSA (§11.5.1), and the challenges used in challenge-response identification systems (§10.3). In all these cases, the quantities generated must be of sufficient size and be “random” in the sense that the probability of any particular value being selected must be sufficiently small to preclude an adversary from gaining advantage through optimizing a search strategy based on such probability. For example, the key space for DES has size 2^{56} . If a secret key k were selected using a true random generator, an adversary would on average have to try 2^{55} possible keys before guessing the correct key k . If, on the other hand, a key k were selected by first choosing a 16-bit random secret s , and then expanding it into a 56-bit key k using a complicated but publicly known function f , the adversary would on average only need to try 2^{15} possible keys (obtained by running every possible value for s through the function f).

This chapter considers techniques for the generation of random and pseudorandom bits and numbers. Related techniques for pseudorandom bit generation that are generally discussed in the literature in the context of stream ciphers, including linear and nonlinear feedback shift registers (Chapter 6) and the output feedback mode (OFB) of block ciphers (Chapter 7), are addressed elsewhere in this book.

Chapter outline

The remainder of §5.1 introduces basic concepts relevant to random and pseudorandom bit generation. §5.2 considers techniques for random bit generation, while §5.3 considers some techniques for pseudorandom bit generation. §5.4 describes statistical tests designed

to measure the quality of a random bit generator. Cryptographically secure pseudorandom bit generators are the topic of §5.5. §5.6 concludes with references and further chapter notes.

5.1.1 Background and Classification

5.1 Definition A *random bit generator* is a device or algorithm which outputs a sequence of statistically independent and unbiased binary digits.

5.2 Remark (*random bits vs. random numbers*) A random bit generator can be used to generate (uniformly distributed) random numbers. For example, a random integer in the interval $[0, n]$ can be obtained by generating a random bit sequence of length $\lceil \lg n \rceil + 1$, and converting it to an integer; if the resulting integer exceeds n , one option is to discard it and generate a new random bit sequence.

§5.2 outlines some physical sources of random bits that are used in practice. Ideally, secrets required in cryptographic algorithms and protocols should be generated with a (true) random bit generator. However, the generation of random bits is an inefficient procedure in most practical environments. Moreover, it may be impractical to securely store and transmit a large number of random bits if these are required in applications such as the one-time pad (§6.1.1). In such situations, the problem can be ameliorated by substituting a random bit generator with a pseudorandom bit generator.

5.3 Definition A *pseudorandom bit generator* (PRBG) is a deterministic¹ algorithm which, given a truly random binary sequence of length k , outputs a binary sequence of length $l \gg k$ which “appears” to be random. The input to the PRBG is called the *seed*, while the output of the PRBG is called a *pseudorandom bit sequence*.

The output of a PRBG is *not* random; in fact, the number of possible output sequences is at most a small fraction, namely $2^k/2^l$, of all possible binary sequences of length l . The intent is to take a small truly random sequence and expand it to a sequence of much larger length, in such a way that an adversary cannot efficiently distinguish between output sequences of the PRBG and truly random sequences of length l . §5.3 discusses ad-hoc techniques for pseudorandom bit generation. In order to gain confidence that such generators are secure, they should be subjected to a variety of statistical tests designed to detect the specific characteristics expected of random sequences. A collection of such tests is given in §5.4. As the following example demonstrates, passing these statistical tests is a *necessary* but not *sufficient* condition for a generator to be secure.

5.4 Example (*linear congruential generators*) A *linear congruential generator* produces a pseudorandom sequence of numbers x_1, x_2, x_3, \dots according to the linear recurrence

$$x_n = ax_{n-1} + b \bmod m, \quad n \geq 1;$$

a , b , and m are *parameters* which characterize the generator, while x_0 is the (secret) *seed*. While such generators are commonly used for simulation purposes and probabilistic algorithms, and pass the statistical tests of §5.4, they are *predictable* and hence entirely insecure for cryptographic purposes: given a partial output sequence, the remainder of the sequence can be reconstructed even if the parameters a , b , and m are unknown. \square

¹*Deterministic* here means that given the same initial seed, the generator will always produce the same output sequence.

A minimum security requirement for a pseudorandom bit generator is that the length k of the random seed should be sufficiently large so that a search over 2^k elements (the total number of possible seeds) is infeasible for the adversary. Two general requirements are that the output sequences of a PRBG should be statistically indistinguishable from truly random sequences, and the output bits should be unpredictable to an adversary with limited computational resources; these requirements are captured in Definitions 5.5 and 5.6.

5.5 Definition A pseudorandom bit generator is said to pass all *polynomial-time² statistical tests* if no polynomial-time algorithm can distinguish between an output sequence of the generator and a truly random sequence of the same length with probability significantly greater than $\frac{1}{2}$.

5.6 Definition A pseudorandom bit generator is said to pass the *next-bit test* if there is no polynomial-time algorithm which, on input of the first l bits of an output sequence s , can predict the $(l + 1)^{\text{st}}$ bit of s with probability significantly greater than $\frac{1}{2}$.

Although Definition 5.5 appears to impose a more stringent security requirement on pseudorandom bit generators than Definition 5.6 does, the next result asserts that they are, in fact, equivalent.

5.7 Fact (*universality of the next-bit test*) A pseudorandom bit generator passes the next-bit test if and only if it passes all polynomial-time statistical tests.

5.8 Definition A PRBG that passes the next-bit test (possibly under some plausible but unproved mathematical assumption such as the intractability of factoring integers) is called a *cryptographically secure pseudorandom bit generator* (CSPRBG).

5.9 Remark (*asymptotic nature of Definitions 5.5, 5.6, and 5.8.*) Each of the three definitions above are given in complexity-theoretic terms and are asymptotic in nature because the notion of “polynomial-time” is meaningful for asymptotically large inputs only; the resulting notions of security are relative in the same sense. In Definitions 5.5, 5.6, 5.8, and Fact 5.7, a pseudorandom bit generator is actually a *family* of such PRBGs. Thus the theoretical security results for a family of PRBGs are only an indirect indication about the security of individual members.

Two cryptographically secure pseudorandom bit generators are presented in §5.5.

5.2 Random bit generation

A (true) random bit generator requires a naturally occurring source of randomness. Designing a hardware device or software program to exploit this randomness and produce a bit sequence that is free of biases and correlations is a difficult task. Additionally, for cryptographic applications, the generator must not be subject to observation or manipulation by an adversary. This section surveys some potential sources of random bits.

Random bit generators based on natural sources of randomness are subject to influence by external factors, and also to malfunction. It is imperative that such devices be tested periodically, for example by using the statistical tests of §5.4.

²The running time of the test is bounded by a polynomial in the length l of the output sequence.

(i) Hardware-based generators

Hardware-based random bit generators exploit the randomness which occurs in some physical phenomena. Such physical processes may produce bits that are biased or correlated, in which case they should be subjected to de-skewing techniques mentioned in (iii) below. Examples of such physical phenomena include:

1. elapsed time between emission of particles during radioactive decay;
2. thermal noise from a semiconductor diode or resistor;
3. the frequency instability of a free running oscillator;
4. the amount a metal insulator semiconductor capacitor is charged during a fixed period of time;
5. air turbulence within a sealed disk drive which causes random fluctuations in disk drive sector read latency times; and
6. sound from a microphone or video input from a camera.

Generators based on the first two phenomena would, in general, have to be built externally to the device using the random bits, and hence may be subject to observation or manipulation by an adversary. Generators based on oscillators and capacitors can be built on VLSI devices; they can be enclosed in tamper-resistant hardware, and hence shielded from active adversaries.

(ii) Software-based generators

Designing a random bit generator in software is even more difficult than doing so in hardware. Processes upon which software random bit generators may be based include:

1. the system clock;
2. elapsed time between keystrokes or mouse movement;
3. content of input/output buffers;
4. user input; and
5. operating system values such as system load and network statistics.

The behavior of such processes can vary considerably depending on various factors, such as the computer platform. It may also be difficult to prevent an adversary from observing or manipulating these processes. For instance, if the adversary has a rough idea of when a random sequence was generated, she can guess the content of the system clock at that time with a high degree of accuracy. A well-designed software random bit generator should utilize as many good sources of randomness as are available. Using many sources guards against the possibility of a few of the sources failing, or being observed or manipulated by an adversary. Each source should be sampled, and the sampled sequences should be combined using a complex *mixing function*; one recommended technique for accomplishing this is to apply a cryptographic hash function such as SHA-1 (Algorithm 9.53) or MD5 (Algorithm 9.51) to a concatenation of the sampled sequences. The purpose of the mixing function is to distill the (true) random bits from the sampled sequences.

(iii) De-skewing

A natural source of random bits may be defective in that the output bits may be *biased* (the probability of the source emitting a 1 is not equal to $\frac{1}{2}$) or *correlated* (the probability of the source emitting a 1 depends on previous bits emitted). There are various techniques for generating truly random bit sequences from the output bits of such a defective generator; such techniques are called *de-skewing techniques*.

5.10 Example (*removing biases in output bits*) Suppose that a generator produces biased but uncorrelated bits. Suppose that the probability of a 1 is p , and the probability of a 0 is $1 - p$, where p is unknown but fixed, $0 < p < 1$. If the output sequence of such a generator is grouped into pairs of bits, with a 10 pair transformed to a 1, a 01 pair transformed to a 0, and 00 and 11 pairs discarded, then the resulting sequence is both unbiased and uncorrelated. \square

A practical (although not provable) de-skewing technique is to pass sequences whose bits are biased or correlated through a cryptographic hash function such as SHA-1 or MD5.

5.3 Pseudorandom bit generation

A one-way function f (Definition 1.12) can be utilized to generate pseudorandom bit sequences (Definition 5.3) by first selecting a random seed s , and then applying the function to the sequence of values $s, s+1, s+2, \dots$; the output sequence is $f(s), f(s+1), f(s+2), \dots$. Depending on the properties of the one-way function used, it may be necessary to only keep a few bits of the output values $f(s+i)$ in order to remove possible correlations between successive values. Examples of suitable one-way functions f include a cryptographic hash function such as SHA-1 (Algorithm 9.53), or a block cipher such as DES (§7.4) with secret key k .

Although such ad-hoc methods have not been proven to be cryptographically secure, they appear sufficient for most applications. Two such methods for pseudorandom bit and number generation which have been standardized are presented in §5.3.1 and §5.3.2. Techniques for the cryptographically secure generation of pseudorandom bits are given in §5.5.

5.3.1 ANSI X9.17 generator

Algorithm 5.11 is a U.S. Federal Information Processing Standard (FIPS) approved method from the ANSI X9.17 standard for the purpose of pseudorandomly generating keys and initialization vectors for use with DES. E_k denotes DES E-D-E two-key triple-encryption (Definition 7.32) under a key k ; the key k should be reserved exclusively for use in this algorithm.

5.11 Algorithm ANSI X9.17 pseudorandom bit generator

INPUT: a random (and secret) 64-bit seed s , integer m , and DES E-D-E encryption key k .

OUTPUT: m pseudorandom 64-bit strings x_1, x_2, \dots, x_m .

1. Compute the intermediate value $I = E_k(D)$, where D is a 64-bit representation of the date/time to as fine a resolution as is available.
2. For i from 1 to m do the following:
 - 2.1 $x_i \leftarrow E_k(I \oplus s)$.
 - 2.2 $s \leftarrow E_k(x_i \oplus I)$.
3. Return(x_1, x_2, \dots, x_m).

Each output bitstring x_i may be used as an initialization vector (IV) for one of the DES modes of operation (§7.2.2). To obtain a DES key from x_i , every eighth bit of x_i should be reset to odd parity (cf. §7.4.2).

5.3.2 FIPS 186 generator

The algorithms presented in this subsection are FIPS-approved methods for pseudorandomly generating the secret parameters for the DSA (Algorithm 11.5.1). Algorithm 5.12 generates a DSA private key a , while Algorithm 5.14 generates the per-message secrets k to be used in signing messages. Both algorithms use a secret seed s which should be randomly generated, and utilize a one-way function constructed by using either SHA-1 (Algorithm 9.53) or DES (Algorithm 7.82), respectively described in Algorithms 5.15 and 5.16.

5.12 Algorithm FIPS 186 pseudorandom number generator for DSA private keys

INPUT: an integer m and a 160-bit prime number q .

OUTPUT: m pseudorandom numbers a_1, a_2, \dots, a_m in the interval $[0, q - 1]$ which may be used as DSA private keys.

1. If Algorithm 5.15 is to be used in step 4.3 then select an arbitrary integer b , $160 \leq b \leq 512$; if Algorithm 5.16 is to be used then set $b \leftarrow 160$.
2. Generate a random (and secret) b -bit seed s .
3. Define the 160-bit string $t = 67452301 \text{ efc dab89 98 badcfe } 10325476 \text{ c3d2e1f0}$ (in hexadecimal).
4. For i from 1 to m do the following:
 - 4.1 (optional user input) Either select a b -bit string y_i , or set $y_i \leftarrow 0$.
 - 4.2 $z_i \leftarrow (s + y_i) \bmod 2^b$.
 - 4.3 $a_i \leftarrow G(t, z_i) \bmod q$. (G is either that defined in Algorithm 5.15 or 5.16.)
 - 4.4 $s \leftarrow (1 + s + a_i) \bmod 2^b$.
5. Return(a_1, a_2, \dots, a_m).

5.13 Note (optional user input) Algorithm 5.12 permits a user to augment the seed s with random or pseudorandom strings derived from alternate sources. The user may desire to do this if she does not trust the quality or integrity of the random bit generator which may be built into a cryptographic module implementing the algorithm.

5.14 Algorithm FIPS 186 pseudorandom number generator for DSA per-message secrets

INPUT: an integer m and a 160-bit prime number q .

OUTPUT: m pseudorandom numbers k_1, k_2, \dots, k_m in the interval $[0, q - 1]$ which may be used as the per-message secret numbers k in the DSA.

1. If Algorithm 5.15 is to be used in step 4.1 then select an integer b , $160 \leq b \leq 512$; if Algorithm 5.16 is to be used then set $b \leftarrow 160$.
2. Generate a random (and secret) b -bit seed s .
3. Define the 160-bit string $t = \text{efcdab89 98 badcfe } 10325476 \text{ c3d2e1f0 } 67452301$ (in hexadecimal).
4. For i from 1 to m do the following:
 - 4.1 $k_i \leftarrow G(t, s) \bmod q$. (G is either that defined in Algorithm 5.15 or 5.16.)
 - 4.2 $s \leftarrow (1 + s + k_i) \bmod 2^b$.
5. Return(k_1, k_2, \dots, k_m).

5.15 Algorithm FIPS 186 one-way function using SHA-1

INPUT: a 160-bit string t and a b -bit string c , $160 \leq b \leq 512$.

OUTPUT: a 160-bit string denoted $G(t, c)$.

1. Break up t into five 32-bit blocks: $t = H_1 \| H_2 \| H_3 \| H_4 \| H_5$.
2. Pad c with 0's to obtain a 512-bit message block: $X \leftarrow c \| 0^{512-b}$.
3. Divide X into 16 32-bit words: $x_0 x_1 \dots x_{15}$, and set $m \leftarrow 1$.
4. Execute step 4 of SHA-1 (Algorithm 9.53). (This alters the H_i 's.)
5. The output is the concatenation: $G(t, c) = H_1 \| H_2 \| H_3 \| H_4 \| H_5$.

5.16 Algorithm FIPS 186 one-way function using DES

INPUT: two 160-bit strings t and c .

OUTPUT: a 160-bit string denoted $G(t, c)$.

1. Break up t into five 32-bit blocks: $t = t_0 \| t_1 \| t_2 \| t_3 \| t_4$.
2. Break up c into five 32-bit blocks: $c = c_0 \| c_1 \| c_2 \| c_3 \| c_4$.
3. For i from 0 to 4 do the following: $x_i \leftarrow t_i \oplus c_i$.
4. For i from 0 to 4 do the following:
 - 4.1 $b_1 \leftarrow c_{(i+4) \bmod 5}$, $b_2 \leftarrow c_{(i+3) \bmod 5}$.
 - 4.2 $a_1 \leftarrow x_i$, $a_2 \leftarrow x_{(i+1) \bmod 5} \oplus x_{(i+4) \bmod 5}$.
 - 4.3 $A \leftarrow a_1 \| a_2$, $B \leftarrow b_1' \| b_2$, where b_1' denotes the 24 least significant bits of b_1 .
 - 4.4 Use DES with key B to encrypt A : $y_i \leftarrow \text{DES}_B(A)$.
 - 4.5 Break up y_i into two 32-bit blocks: $y_i = L_i \| R_i$.
5. For i from 0 to 4 do the following: $z_i \leftarrow L_i \oplus R_{(i+2) \bmod 5} \oplus L_{(i+3) \bmod 5}$.
6. The output is the concatenation: $G(t, c) = z_0 \| z_1 \| z_2 \| z_3 \| z_4$.

5.4 Statistical tests

This section presents some tests designed to measure the quality of a generator purported to be a random bit generator (Definition 5.1). While it is impossible to give a mathematical proof that a generator is indeed a random bit generator, the tests described here help detect certain kinds of weaknesses the generator may have. This is accomplished by taking a sample output sequence of the generator and subjecting it to various statistical tests. Each statistical test determines whether the sequence possesses a certain attribute that a truly random sequence would be likely to exhibit; the conclusion of each test is not definite, but rather *probabilistic*. An example of such an attribute is that the sequence should have roughly the same number of 0's as 1's. If the sequence is deemed to have failed any one of the statistical tests, the generator may be *rejected* as being non-random; alternatively, the generator may be subjected to further testing. On the other hand, if the sequence passes all of the statistical tests, the generator is *accepted* as being random. More precisely, the term "accepted" should be replaced by "not rejected", since passing the tests merely provides probabilistic evidence that the generator produces sequences which have certain characteristics of random sequences.

§5.4.1 and §5.4.2 provide some relevant background in statistics. §5.4.3 establishes some notation and lists Golomb's randomness postulates. Specific statistical tests for randomness are described in §5.4.4 and §5.4.5.

5.4.1 The normal and chi-square distributions

The normal and χ^2 distributions are widely used in statistical applications.

5.17 Definition If the result X of an experiment can be any real number, then X is said to be a *continuous* random variable.

5.18 Definition A *probability density function* of a continuous random variable X is a function $f(x)$ which can be integrated and satisfies:

- (i) $f(x) \geq 0$ for all $x \in \mathbb{R}$;
- (ii) $\int_{-\infty}^{\infty} f(x) dx = 1$; and
- (iii) for all $a, b \in \mathbb{R}$, $P(a < X \leq b) = \int_a^b f(x) dx$.

(i) The normal distribution

The normal distribution arises in practice when a large number of independent random variables having the same mean and variance are summed.

5.19 Definition A (continuous) random variable X has a *normal distribution* with mean μ and variance σ^2 if its probability density function is defined by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp \left\{ -\frac{(x-\mu)^2}{2\sigma^2} \right\}, \quad -\infty < x < \infty.$$

Notation: X is said to be $N(\mu, \sigma^2)$. If X is $N(0, 1)$, then X is said to have a *standard normal distribution*.

A graph of the $N(0, 1)$ distribution is given in Figure 5.1. The graph is symmetric

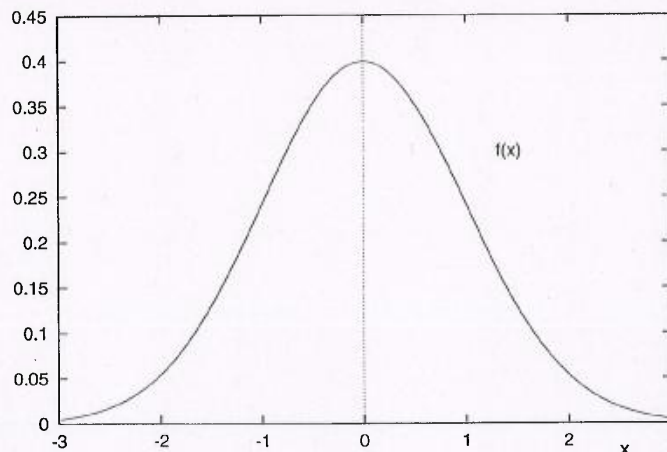


Figure 5.1: The normal distribution $N(0, 1)$.

about the vertical axis, and hence $P(X > x) = P(X < -x)$ for any x . Table 5.1 gives some percentiles for the standard normal distribution. For example, the entry ($\alpha = 0.05$, $x = 1.6449$) means that if X is $N(0, 1)$, then X exceeds 1.6449 about 5% of the time.

Fact 5.20 can be used to reduce questions about a normal distribution to questions about the standard normal distribution.

α	0.1	0.05	0.025	0.01	0.005	0.0025	0.001	0.0005
x	1.2816	1.6449	1.9600	2.3263	2.5758	2.8070	3.0902	3.2905

Table 5.1: Selected percentiles of the standard normal distribution. If X is a random variable having a standard normal distribution, then $P(X > x) = \alpha$.

5.20 Fact If the random variable X is $N(\mu, \sigma^2)$, then the random variable $Z = (X - \mu)/\sigma$ is $N(0, 1)$.

(ii) The χ^2 distribution

The χ^2 distribution can be used to compare the *goodness-of-fit* of the observed frequencies of events to their expected frequencies under a hypothesized distribution. The χ^2 distribution with v degrees of freedom arises in practice when the squares of v independent random variables having standard normal distributions are summed.

5.21 Definition A (continuous) random variable X has a χ^2 (*chi-square*) distribution with v degrees of freedom if its probability density function is defined by

$$f(x) = \begin{cases} \frac{1}{\Gamma(v/2)2^{v/2}} x^{(v/2)-1} e^{-x/2}, & 0 \leq x < \infty, \\ 0, & x < 0, \end{cases}$$

where Γ is the gamma function.³ The mean and variance of this distribution are $\mu = v$, and $\sigma^2 = 2v$.

A graph of the χ^2 distribution with $v = 7$ degrees of freedom is given in Figure 5.2. Table 5.2 gives some percentiles of the χ^2 distribution for various degrees of freedom. For

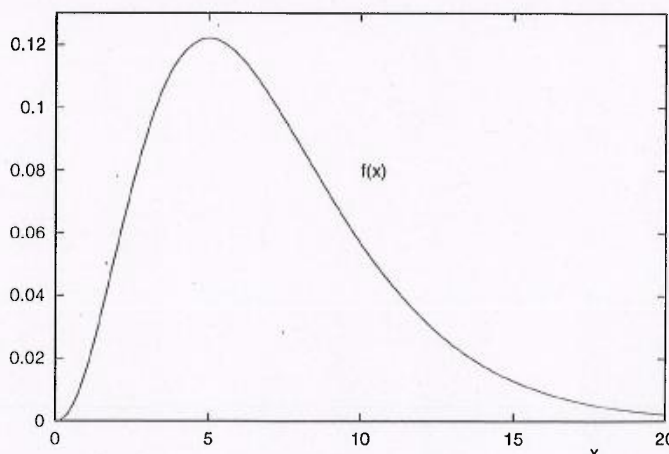


Figure 5.2: The χ^2 (*chi-square*) distribution with $v = 7$ degrees of freedom.

example, the entry in row $v = 5$ and column $\alpha = 0.05$ is $x = 11.0705$; this means that if X has a χ^2 distribution with 5 degrees of freedom, then X exceeds 11.0705 about 5% of the time.

³The gamma function is defined by $\Gamma(t) = \int_0^\infty x^{t-1} e^{-x} dx$, for $t > 0$.

v	α					
	0.100	0.050	0.025	0.010	0.005	0.001
1	2.7055	3.8415	5.0239	6.6349	7.8794	10.8276
2	4.6052	5.9915	7.3778	9.2103	10.5966	13.8155
3	6.2514	7.8147	9.3484	11.3449	12.8382	16.2662
4	7.7794	9.4877	11.1433	13.2767	14.8603	18.4668
5	9.2364	11.0705	12.8325	15.0863	16.7496	20.5150
6	10.6446	12.5916	14.4494	16.8119	18.5476	22.4577
7	12.0170	14.0671	16.0128	18.4753	20.2777	24.3219
8	13.3616	15.5073	17.5345	20.0902	21.9550	26.1245
9	14.6837	16.9190	19.0228	21.6660	23.5894	27.8772
10	15.9872	18.3070	20.4832	23.2093	25.1882	29.5883
11	17.2750	19.6751	21.9200	24.7250	26.7568	31.2641
12	18.5493	21.0261	23.3367	26.2170	28.2995	32.9095
13	19.8119	22.3620	24.7356	27.6882	29.8195	34.5282
14	21.0641	23.6848	26.1189	29.1412	31.3193	36.1233
15	22.3071	24.9958	27.4884	30.5779	32.8013	37.6973
16	23.5418	26.2962	28.8454	31.9999	34.2672	39.2524
17	24.7690	27.5871	30.1910	33.4087	35.7185	40.7902
18	25.9894	28.8693	31.5264	34.8053	37.1565	42.3124
19	27.2036	30.1435	32.8523	36.1909	38.5823	43.8202
20	28.4120	31.4104	34.1696	37.5662	39.9968	45.3147
21	29.6151	32.6706	35.4789	38.9322	41.4011	46.7970
22	30.8133	33.9244	36.7807	40.2894	42.7957	48.2679
23	32.0069	35.1725	38.0756	41.6384	44.1813	49.7282
24	33.1962	36.4150	39.3641	42.9798	45.5585	51.1786
25	34.3816	37.6525	40.6465	44.3141	46.9279	52.6197
26	35.5632	38.8851	41.9232	45.6417	48.2899	54.0520
27	36.7412	40.1133	43.1945	46.9629	49.6449	55.4760
28	37.9159	41.3371	44.4608	48.2782	50.9934	56.8923
29	39.0875	42.5570	45.7223	49.5879	52.3356	58.3012
30	40.2560	43.7730	46.9792	50.8922	53.6720	59.7031
31	41.4217	44.9853	48.2319	52.1914	55.0027	61.0983
63	77.7454	82.5287	86.8296	92.0100	95.6493	103.4424
127	147.8048	154.3015	160.0858	166.9874	171.7961	181.9930
255	284.3359	293.2478	301.1250	310.4574	316.9194	330.5197
511	552.3739	564.6961	575.5298	588.2978	597.0978	615.5149
1023	1081.3794	1098.5208	1113.5334	1131.1587	1143.2653	1168.4972

Table 5.2: Selected percentiles of the χ^2 (chi-square) distribution. A (v, α) -entry of x in the table has the following meaning: if X is a random variable having a χ^2 distribution with v degrees of freedom, then $P(X > x) = \alpha$.

Fact 5.22 relates the normal distribution to the χ^2 distribution.

5.22 Fact If the random variable X is $N(\mu, \sigma^2)$, $\sigma^2 > 0$, then the random variable $Z = (X - \mu)^2/\sigma^2$ has a χ^2 distribution with 1 degree of freedom. In particular, if X is $N(0, 1)$, then $Z = X^2$ has a χ^2 distribution with 1 degree of freedom.

5.4.2 Hypothesis testing

A *statistical hypothesis*, denoted H_0 , is an assertion about a distribution of one or more random variables. A *test* of a statistical hypothesis is a procedure, based upon observed values of the random variables, that leads to the acceptance or rejection of the hypothesis H_0 . The test only provides a measure of the strength of the evidence provided by the data against the hypothesis; hence, the conclusion of the test is not definite, but rather probabilistic.

5.23 Definition The *significance level* α of the test of a statistical hypothesis H_0 is the probability of rejecting H_0 when it is true.

In this section, H_0 will be the hypothesis that a given binary sequence was produced by a random bit generator. If the significance level α of a test of H_0 is too high, then the test may reject sequences that were, in fact, produced by a random bit generator (such an error is called a *Type I error*). On the other hand, if the significance level of a test of H_0 is too low, then there is the danger that the test may accept sequences even though they were not produced by a random bit generator (such an error is called a *Type II error*).⁴ It is, therefore, important that the test be carefully designed to have a significance level that is appropriate for the purpose at hand; a significance level α between 0.001 and 0.05 might be employed in practice.

A statistical test is implemented by specifying a *statistic* on the random sample.⁵ Statistics are generally chosen so that they can be efficiently computed, and so that they (approximately) follow an $N(0, 1)$ or a χ^2 distribution (see §5.4.1). The value of the statistic for the sample output sequence is computed and compared with the value expected for a random sequence as described below.

1. Suppose that a statistic X for a random sequence follows a χ^2 distribution with v degrees of freedom, and suppose that the statistic can be expected to take on larger values for nonrandom sequences. To achieve a significance level of α , a *threshold* value x_α is chosen (using Table 5.2) so that $P(X > x_\alpha) = \alpha$. If the value X_s of the statistic for the sample output sequence satisfies $X_s > x_\alpha$, then the sequence *fails* the test; otherwise, it *passes* the test. Such a test is called a *one-sided* test. For example, if $v = 5$ and $\alpha = 0.025$, then $x_\alpha = 12.8325$, and one expects a random sequence to fail the test only 2.5% of the time.
2. Suppose that a statistic X for a random sequence follows an $N(0, 1)$ distribution, and suppose that the statistic can be expected to take on both larger and smaller values for nonrandom sequences. To achieve a significance level of α , a *threshold* value x_α is chosen (using Table 5.1) so that $P(X > x_\alpha) = P(X < -x_\alpha) = \alpha/2$. If the value

⁴Actually, the probability β of a Type II error may be completely independent of α . If the generator is not a random bit generator, the probability β depends on the nature of the defects of the generator, and is usually difficult to determine in practice. For this reason, assuming that the probability of a Type II error is proportional to α is a useful intuitive guide when selecting an appropriate significance level for a test.

⁵A *statistic* is a function of the elements of a random sample; for example, the number of 0's in a binary sequence is a statistic.

X_s of the statistic for the sample output sequence satisfies $X_s > x_\alpha$ or $X_s < -x_\alpha$, then the sequence *fails* the test; otherwise, it *passes* the test. Such a test is called a *two-sided* test. For example, if $\alpha = 0.05$, then $x_\alpha = 1.96$, and one expects a random sequence to fail the test only 5% of the time.

5.4.3 Golomb's randomness postulates

Golomb's randomness postulates (Definition 5.28) are presented here for historical reasons – they were one of the first attempts to establish some *necessary* conditions for a periodic pseudorandom sequence to look random. It is emphasized that these conditions are far from being *sufficient* for such sequences to be considered random. Unless otherwise stated, all sequences are binary sequences.

5.24 Definition Let $s = s_0, s_1, s_2, \dots$ be an infinite sequence. The subsequence consisting of the first n terms of s is denoted by $s^n = s_0, s_1, \dots, s_{n-1}$.

5.25 Definition The sequence $s = s_0, s_1, s_2, \dots$ is said to be *N-periodic* if $s_i = s_{i+N}$ for all $i \geq 0$. The sequence s is *periodic* if it is *N-periodic* for some positive integer N . The *period* of a periodic sequence s is the smallest positive integer N for which s is *N-periodic*. If s is a periodic sequence of period N , then the *cycle* of s is the subsequence s^N .

5.26 Definition Let s be a sequence. A *run* of s is a subsequence of s consisting of consecutive 0's or consecutive 1's which is neither preceded nor succeeded by the same symbol. A run of 0's is called a *gap*, while a run of 1's is called a *block*.

5.27 Definition Let $s = s_0, s_1, s_2, \dots$ be a periodic sequence of period N . The *autocorrelation function* of s is the integer-valued function $C(t)$ defined as

$$C(t) = \frac{1}{N} \sum_{i=0}^{N-1} (2s_i - 1) \cdot (2s_{i+t} - 1), \quad \text{for } 0 \leq t \leq N-1.$$

The autocorrelation function $C(t)$ measures the amount of similarity between the sequence s and a shift of s by t positions. If s is a random periodic sequence of period N , then $|N \cdot C(t)|$ can be expected to be quite small for all values of t , $0 < t < N$.

5.28 Definition Let s be a periodic sequence of period N . *Golomb's randomness postulates* are the following.

- R1: In the cycle s^N of s , the number of 1's differs from the number of 0's by at most 1.
- R2: In the cycle s^N , at least half the runs have length 1, at least one-fourth have length 2, at least one-eighth have length 3, etc., as long as the number of runs so indicated exceeds 1. Moreover, for each of these lengths, there are (almost) equally many gaps and blocks.⁶
- R3: The autocorrelation function $C(t)$ is two-valued. That is for some integer K ,

$$N \cdot C(t) = \sum_{i=0}^{N-1} (2s_i - 1) \cdot (2s_{i+t} - 1) = \begin{cases} N, & \text{if } t = 0, \\ K, & \text{if } 1 \leq t \leq N-1. \end{cases}$$

⁶Postulate R2 implies postulate R1.

5.29 Definition A binary sequence which satisfies Golomb's randomness postulates is called a *pseudo-noise sequence* or a *pn-sequence*.

Pseudo-noise sequences arise in practice as output sequences of maximum-length linear feedback shift registers (cf. Fact 6.14).

5.30 Example (pn-sequence) Consider the periodic sequence s of period $N = 15$ with cycle

$$s^{15} = 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1.$$

The following shows that the sequence s satisfies Golomb's randomness postulates.

R1: The number of 0's in s^{15} is 7, while the number of 1's is 8.

R2: s^{15} has 8 runs. There are 4 runs of length 1 (2 gaps and 2 blocks), 2 runs of length 2 (1 gap and 1 block), 1 run of length 3 (1 gap), and 1 run of length 4 (1 block).

R3: The autocorrelation function $C(t)$ takes on two values: $C(0) = 1$ and $C(t) = -\frac{1}{15}$ for $1 \leq t \leq 14$.

Hence, s is a pn-sequence. □

5.4.4 Five basic tests

Let $s = s_0, s_1, s_2, \dots, s_{n-1}$ be a binary sequence of length n . This subsection presents five statistical tests that are commonly used for determining whether the binary sequence s possesses some specific characteristics that a truly random sequence would be likely to exhibit. It is emphasized again that the outcome of each test is not definite, but rather probabilistic. If a sequence passes all five tests, there is no guarantee that it was indeed produced by a random bit generator (cf. Example 5.4).

(i) Frequency test (monobit test)

The purpose of this test is to determine whether the number of 0's and 1's in s are approximately the same, as would be expected for a random sequence. Let n_0, n_1 denote the number of 0's and 1's in s , respectively. The statistic used is

$$X_1 = \frac{(n_0 - n_1)^2}{n} \quad (5.1)$$

which approximately follows a χ^2 distribution with 1 degree of freedom if $n \geq 10$.⁷

(ii) Serial test (two-bit test)

The purpose of this test is to determine whether the number of occurrences of 00, 01, 10, and 11 as subsequences of s are approximately the same, as would be expected for a random sequence. Let n_0, n_1 denote the number of 0's and 1's in s , respectively, and let $n_{00}, n_{01}, n_{10}, n_{11}$ denote the number of occurrences of 00, 01, 10, 11 in s , respectively. Note that $n_{00} + n_{01} + n_{10} + n_{11} = (n - 1)$ since the subsequences are allowed to overlap. The statistic used is

$$X_2 = \frac{4}{n-1} (n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2) - \frac{2}{n} (n_0^2 + n_1^2) + 1 \quad (5.2)$$

which approximately follows a χ^2 distribution with 2 degrees of freedom if $n \geq 21$.

⁷In practice, it is recommended that the length n of the sample output sequence be much larger (for example, $n \gg 10000$) than the minimum specified for each test in this subsection.

(iii) Poker test

Let m be a positive integer such that $\lfloor \frac{n}{m} \rfloor \geq 5 \cdot (2^m)$, and let $k = \lfloor \frac{n}{m} \rfloor$. Divide the sequence s into k non-overlapping parts each of length m , and let n_i be the number of occurrences of the i^{th} type of sequence of length m , $1 \leq i \leq 2^m$. The poker test determines whether the sequences of length m each appear approximately the same number of times in s , as would be expected for a random sequence. The statistic used is

$$X_3 = \frac{2^m}{k} \left(\sum_{i=1}^{2^m} n_i^2 \right) - k \quad (5.3)$$

which approximately follows a χ^2 distribution with $2^m - 1$ degrees of freedom. Note that the poker test is a generalization of the frequency test: setting $m = 1$ in the poker test yields the frequency test.

(iv) Runs test

The purpose of the runs test is to determine whether the number of runs (of either zeros or ones; see Definition 5.26) of various lengths in the sequence s is as expected for a random sequence. The expected number of gaps (or blocks) of length i in a random sequence of length n is $e_i = (n - i + 3)/2^{i+2}$. Let k be equal to the largest integer i for which $e_i \geq 5$. Let B_i, G_i be the number of blocks and gaps, respectively, of length i in s for each i , $1 \leq i \leq k$. The statistic used is

$$X_4 = \sum_{i=1}^k \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^k \frac{(G_i - e_i)^2}{e_i} \quad (5.4)$$

which approximately follows a χ^2 distribution with $2k - 2$ degrees of freedom.

(v) Autocorrelation test

The purpose of this test is to check for correlations between the sequence s and (non-cyclic) shifted versions of it. Let d be a fixed integer, $1 \leq d \leq \lfloor n/2 \rfloor$. The number of bits in s not equal to their d -shifts is $A(d) = \sum_{i=0}^{n-d-1} s_i \oplus s_{i+d}$, where \oplus denotes the XOR operator. The statistic used is

$$X_5 = 2 \left(A(d) - \frac{n-d}{2} \right) / \sqrt{n-d} \quad (5.5)$$

which approximately follows an $N(0, 1)$ distribution if $n - d \geq 10$. Since small values of $A(d)$ are as unexpected as large values of $A(d)$, a two-sided test should be used.

5.31 Example (basic statistical tests) Consider the (non-random) sequence s of length $n = 160$ obtained by replicating the following sequence four times:

11100 01100 01000 10100 11101 11100 10010 01001.

- (i) (frequency test) $n_0 = 84$, $n_1 = 76$, and the value of the statistic X_1 is 0.4.
- (ii) (serial test) $n_{00} = 44$, $n_{01} = 40$, $n_{10} = 40$, $n_{11} = 35$, and the value of the statistic X_2 is 0.6252.
- (iii) (poker test) Here $m = 3$ and $k = 53$. The blocks 000, 001, 010, 011, 100, 101, 110, 111 appear 5, 10, 6, 4, 12, 3, 6, and 7 times, respectively, and the value of the statistic X_3 is 9.6415.
- (iv) (runs test) Here $e_1 = 20.25$, $e_2 = 10.0625$, $e_3 = 5$, and $k = 3$. There are 25, 4, 5 blocks of lengths 1, 2, 3, respectively, and 8, 20, 12 gaps of lengths 1, 2, 3, respectively. The value of the statistic X_4 is 31.7913.

(v) (*autocorrelation test*) If $d = 3$, then $A(3) = 35$. The value of the statistic X_5 is -6.9434 .

For a significance level of $\alpha = 0.05$, the threshold values for X_1, X_2, X_3, X_4 , and X_5 are 3.8415, 5.9915, 14.0671, 9.4877, and 1.96, respectively (see Tables 5.1 and 5.2). Hence, the given sequence s passes the frequency, serial, and poker tests, but fails the runs and autocorrelation tests. \square

5.32 Note (FIPS 140-1 statistical tests for randomness) FIPS 140-1 specifies four statistical tests for randomness. Instead of making the user select appropriate significance levels for these tests, explicit bounds are provided that the computed value of a statistic must satisfy. A single bitstring s of length 20000 bits, output from a generator, is subjected to each of the following tests. If any of the tests fail, then the generator fails the test.

- (i) *monobit test*. The number n_1 of 1's in s should satisfy $9654 < n_1 < 10346$.
- (ii) *poker test*. The statistic X_3 defined by equation (5.3) is computed for $m = 4$. The poker test is passed if $1.03 < X_3 < 57.4$.
- (iii) *runs test*. The number B_i and G_i of blocks and gaps, respectively, of length i in s are counted for each i , $1 \leq i \leq 6$. (For the purpose of this test, runs of length greater than 6 are considered to be of length 6.) The runs test is passed if the 12 counts $B_i, G_i, 1 \leq i \leq 6$, are each within the corresponding interval specified by the following table.

Length of run	Required interval
1	2267 – 2733
2	1079 – 1421
3	502 – 748
4	223 – 402
5	90 – 223
6	90 – 223

- (iv) *long run test*. The long run test is passed if there are no runs of length 34 or more.

For high security applications, FIPS 140-1 mandates that the four tests be performed each time the random bit generator is powered up. FIPS 140-1 allows these tests to be substituted by alternative tests which provide equivalent or superior randomness checking.

5.4.5 Maurer's universal statistical test

The basic idea behind Maurer's universal statistical test is that it should not be possible to significantly compress (without loss of information) the output sequence of a random bit generator. Thus, if a sample output sequence s of a bit generator can be significantly compressed, the generator should be rejected as being defective. Instead of actually compressing the sequence s , the universal statistical test computes a quantity that is related to the length of the compressed sequence.

The *universality* of Maurer's universal statistical test arises because it is able to detect any one of a very general class of possible defects a bit generator might have. This class includes the five defects that are detectable by the basic tests of §5.4.4. A drawback of the universal statistical test over the five basic tests is that it requires a much longer sample output sequence in order to be effective. Provided that the required output sequence can be efficiently generated, this drawback is not a practical concern since the universal statistical test itself is very efficient.

Algorithm 5.33 computes the statistic X_u for a sample output sequence $s = s_0, s_1, \dots, s_{n-1}$ to be used in the universal statistical test. The parameter L is first chosen from the

L	μ	σ_1^2
1	0.7326495	0.690
2	1.5374383	1.338
3	2.4016068	1.901
4	3.3112247	2.358
5	4.2534266	2.705
6	5.2177052	2.954
7	6.1962507	3.125
8	7.1836656	3.238

L	μ	σ_1^2
9	8.1764248	3.311
10	9.1723243	3.356
11	10.170032	3.384
12	11.168765	3.401
13	12.168070	3.410
14	13.167693	3.416
15	14.167488	3.419
16	15.167379	3.421

Table 5.3: Mean μ and variance σ^2 of the statistic X_u for random sequences, with parameters L, K as $Q \rightarrow \infty$. The variance of X_u is $\sigma^2 = c(L, K)^2 \cdot \sigma_1^2 / K$, where $c(L, K) \approx 0.7 - (0.8/L) + (1.6 + (12.8/L)) \cdot K^{-4/L}$ for $K \geq 2^L$.

interval $[6, 16]$. The sequence s is then partitioned into non-overlapping L -bit blocks, with any leftover bits discarded; the total number of blocks is $Q + K$, where Q and K are defined below. For each i , $1 \leq i \leq Q + K$, let b_i be the integer whose binary representation is the i^{th} block. The blocks are scanned in order. A table T is maintained so that at each stage $T[j]$ is the position of the last occurrence of the block corresponding to integer j , $0 \leq j \leq 2^L - 1$. The first Q blocks of s are used to initialize table T ; Q should be chosen to be at least $10 \cdot 2^L$ in order to have a high likelihood that each of the 2^L L -bit blocks occurs at least once in the first Q blocks. The remaining K blocks are used to define the statistic X_u as follows. For each i , $Q + 1 \leq i \leq Q + K$, let $A_i = i - T[b_i]$; A_i is the number of positions since the last occurrence of block b_i . Then

$$X_u = \frac{1}{K} \sum_{i=Q+1}^{Q+K} \lg A_i. \quad (5.6)$$

K should be at least $1000 \cdot 2^L$ (and, hence, the sample sequence s should be at least $(1010 \cdot 2^L \cdot L)$ bits in length). Table 5.3 lists the mean μ and variance σ^2 of X_u for random sequences for some sample choices of L as $Q \rightarrow \infty$.

5.33 Algorithm Computing the statistic X_u for Maurer's universal statistical test

INPUT: a binary sequence $s = s_0, s_1, \dots, s_{n-1}$ of length n , and parameters L, Q, K .

OUTPUT: the value of the statistic X_u for the sequence s .

1. Zero the table T . For j from 0 to $2^L - 1$ do the following: $T[j] \leftarrow 0$.
2. Initialize the table T . For i from 1 to Q do the following: $T[b_i] \leftarrow i$.
3. $\text{sum} \leftarrow 0$.
4. For i from $Q + 1$ to $Q + K$ do the following:
 - 4.1 $\text{sum} \leftarrow \text{sum} + \lg(i - T[b_i])$.
 - 4.2 $T[b_i] \leftarrow i$.
5. $X_u \leftarrow \text{sum} / K$.
6. Return(X_u).

Maurer's universal statistical test uses the computed value of X_u for the sample output sequence s in the manner prescribed by Fact 5.34. To test the sequence s , a two-sided test should be used with a significance level α between 0.001 and 0.01 (see §5.4.2).

5.34 Fact Let X_u be the statistic defined in (5.6) having mean μ and variance σ^2 as given in Table 5.3. Then, for random sequences, the statistic $Z_u = (X_u - \mu)/\sigma$ approximately follows an $N(0, 1)$ distribution.

5.5 Cryptographically secure pseudorandom bit generation

Two cryptographically secure pseudorandom bit generators (CSPRNG – see Definition 5.8) are presented in this section. The security of each generator relies on the presumed intractability of an underlying number-theoretic problem. The modular multiplications that these generators use make them relatively slow compared to the (ad-hoc) pseudorandom bit generators of §5.3. Nevertheless they may be useful in some circumstances, for example, generating pseudorandom bits on hardware devices which already have the circuitry for performing modular multiplications. Efficient techniques for implementing modular multiplication are presented in §14.3.

5.5.1 RSA pseudorandom bit generator

The RSA pseudorandom bit generator is a CSPRNG under the assumption that the RSA problem is intractable (§3.3; see also §3.9.2).

5.35 Algorithm RSA pseudorandom bit generator

SUMMARY: a pseudorandom bit sequence z_1, z_2, \dots, z_l of length l is generated.

1. *Setup.* Generate two secret RSA-like primes p and q (cf. Note 8.8), and compute $n = pq$ and $\phi = (p-1)(q-1)$. Select a random integer e , $1 < e < \phi$, such that $\gcd(e, \phi) = 1$.
2. Select a random integer x_0 (the *seed*) in the interval $[1, n-1]$.
3. For i from 1 to l do the following:
 - 3.1 $x_i \leftarrow x_{i-1}^e \bmod n$.
 - 3.2 $z_i \leftarrow$ the least significant bit of x_i .
4. The output sequence is z_1, z_2, \dots, z_l .

5.36 Note (*efficiency of the RSA PRBG*) If $e = 3$ is chosen (cf. Note 8.9(ii)), then generating each pseudorandom bit z_i requires one modular multiplication and one modular squaring. The efficiency of the generator can be improved by extracting the j least significant bits of x_i in step 3.2, where $j = c \lg \lg n$ and c is a constant. Provided that n is sufficiently large, this modified generator is also cryptographically secure (cf. Fact 3.87). For a modulus n of a fixed bitlength (e.g., 1024 bits), an explicit range of values of c for which the resulting generator remains cryptographically secure (cf. Remark 5.9) under the intractability assumption of the RSA problem has not been determined.

The following modification improves the efficiency of the RSA PRBG.

5.37 Algorithm Micali-Schnorr pseudorandom bit generator

SUMMARY: a pseudorandom bit sequence is generated.

1. *Setup.* Generate two secret RSA-like primes p and q (cf. Note 8.8), and compute $n = pq$ and $\phi = (p-1)(q-1)$. Let $N = \lfloor \lg n \rfloor + 1$ (the bitlength of n). Select an integer e , $1 < e < \phi$, such that $\gcd(e, \phi) = 1$ and $80e \leq N$. Let $k = \lfloor N(1 - \frac{2}{e}) \rfloor$ and $r = N - k$.
2. Select a random sequence x_0 (the *seed*) of bitlength r .
3. *Generate a pseudorandom sequence of length $k \cdot l$.* For i from 1 to l do the following:
 - 3.1 $y_i \leftarrow x_{i-1}^e \bmod n$.
 - 3.2 $x_i \leftarrow$ the r most significant bits of y_i .
 - 3.3 $z_i \leftarrow$ the k least significant bits of y_i .
4. The output sequence is $z_1 \parallel z_2 \parallel \dots \parallel z_l$, where \parallel denotes concatenation.

5.38 Note (*efficiency of the Micali-Schnorr PRBG*) Algorithm 5.37 is more efficient than the RSA PRBG since $\lfloor N(1 - \frac{2}{e}) \rfloor$ bits are generated per exponentiation by e . For example, if $e = 3$ and $N = 1024$, then $k = 341$ bits are generated per exponentiation. Moreover, each exponentiation requires only one modular squaring of an $r = 683$ -bit number, and one modular multiplication.

5.39 Note (*security of the Micali-Schnorr PRBG*) Algorithm 5.37 is cryptographically secure under the assumption that the following is true: the distribution $x^e \bmod n$ for random r -bit sequences x is indistinguishable by all polynomial-time statistical tests from the uniform distribution of integers in the interval $[0, n-1]$. This assumption is stronger than requiring that the RSA problem be intractable.

5.5.2 Blum-Blum-Shub pseudorandom bit generator

The Blum-Blum-Shub pseudorandom bit generator (also known as the $x^2 \bmod n$ generator or the *BBS* generator) is a CSPRBG under the assumption that integer factorization is intractable (§3.2). It forms the basis for the Blum-Goldwasser probabilistic public-key encryption scheme (Algorithm 8.55).

5.40 Algorithm Blum-Blum-Shub pseudorandom bit generator

SUMMARY: a pseudorandom bit sequence z_1, z_2, \dots, z_l of length l is generated.

1. *Setup.* Generate two large secret random (and distinct) primes p and q (cf. Note 8.8), each congruent to 3 modulo 4, and compute $n = pq$.
2. Select a random integer s (the *seed*) in the interval $[1, n-1]$ such that $\gcd(s, n) = 1$, and compute $x_0 \leftarrow s^2 \bmod n$.
3. For i from 1 to l do the following:
 - 3.1 $x_i \leftarrow x_{i-1}^2 \bmod n$.
 - 3.2 $z_i \leftarrow$ the least significant bit of x_i .
4. The output sequence is z_1, z_2, \dots, z_l .

- 5.41 Note** (*efficiency of the Blum-Blum-Shub PRBG*) Generating each pseudorandom bit z_i requires one modular squaring. The efficiency of the generator can be improved by extracting the j least significant bits of x_i in step 3.2, where $j = c \lg \lg n$ and c is a constant. Provided that n is sufficiently large, this modified generator is also cryptographically secure. For a modulus n of a fixed bitlength (eg. 1024 bits), an explicit range of values of c for which the resulting generator is cryptographically secure (cf. Remark 5.9) under the intractability assumption of the integer factorization problem has not been determined.

5.6 Notes and further references

§5.1

Chapter 3 of Knuth [692] is the definitive reference for the classic (non-cryptographic) generation of pseudorandom numbers. Knuth [692, pp.142-166] contains an extensive discussion of what it means for a sequence to be random. Lagarias [724] gives a survey of theoretical results on pseudorandom number generators. Luby [774] provides a comprehensive and rigorous overview of pseudorandom generators.

For a study of linear congruential generators (Example 5.4), see Knuth [692, pp.9-25]. Plumstead/Boyar [979, 980] showed how to predict the output of a linear congruential generator given only a few elements of the output sequence, and when the parameters a , b , and m of the generator are unknown. Boyar [180] extended her method and showed that linear *multivariate* congruential generators (having recurrence equation $x_n = a_1 x_{n-1} + a_2 x_{n-2} + \dots + a_l x_{n-l} + b \bmod m$), and *quadratic* congruential generators (having recurrence equation $x_n = ax_{n-1}^2 + bx_{n-1} + c \bmod m$) are cryptographically insecure. Finally, Krawczyk [713] generalized these results and showed how the output of any *multivariate polynomial* congruential generator can be efficiently predicted. A *truncated* linear congruential generator is one where a fraction of the least significant bits of the x_i are discarded. Frieze et al. [427] showed that these generators can be efficiently predicted if the generator parameters a , b , and m are known. Stern [1173] extended this method to the case where only m is known. Boyar [179] presented an efficient algorithm for predicting linear congruential generators when $O(\log \log m)$ bits are discarded, and when the parameters a , b , and m are unknown. No efficient prediction algorithms are known for truncated multivariate polynomial congruential generators. For a summary of cryptanalytic attacks on congruential generators, see Brickell and Odlyzko [209, pp.523-526].

For a formal definition of a statistical test (Definition 5.5), see Yao [1258]. Fact 5.7 on the universality of the next-bit test is due to Yao [1258]. For a proof of Yao's result, see Kranakis [710] and §12.2 of Stinson [1178]. A proof of a generalization of Yao's result is given by Goldreich, Goldwasser, and Micali [468]. The notion of a cryptographically secure pseudorandom bit generator (Definition 5.8) was introduced by Blum and Micali [166]. Blum and Micali also gave a formal description of the next-bit test (Definition 5.6), and presented the first cryptographically secure pseudorandom bit generator whose security is based on the discrete logarithm problem (see page 189). Universal tests were presented by Schifft and Shamir [1103] for verifying the assumed properties of a pseudorandom generator whose output sequences are not necessarily uniformly distributed.

The first provably secure pseudorandom *number* generator was proposed by Shamir [1112]. Shamir proved that predicting the next number of an output sequence of this generator is equivalent to inverting the RSA function. However, even though the numbers as a whole may be unpredictable, certain parts of the number (for example, its least significant bit) may

be biased or predictable. Hence, Shamir's generator is not cryptographically secure in the sense of Definition 5.8.

§5.2

Agnew [17] proposed a VLSI implementation of a random bit generator consisting of two identical metal insulator semiconductor capacitors close to each other. The cells are charged over the same period of time, and then a 1 or 0 is assigned depending on which cell has a greater charge. Fairfield, Mortenson, and Coulthart [382] described an LSI random bit generator based on the frequency instability of a free running oscillator. Davis, Ihaka, and Fenstermacher [309] used the unpredictability of air turbulence occurring in a sealed disk drive as a random bit generator. The bits are extracted by measuring the variations in the time to access disk blocks. Fast Fourier Transform (FFT) techniques are then used to remove possible biases and correlations. A sample implementation generated 100 random bits per minute. For further guidance on hardware and software-based techniques for generating random bits, see RFC 1750 [1043].

The de-skewing technique of Example 5.10 is due to von Neumann [1223]. Elias [370] generalized von Neumann's technique to a more efficient scheme (one where fewer bits are discarded). Fast Fourier Transform techniques for removing biases and correlations are described by Brillinger [213]. For further ways of removing correlations, see Blum [161], Santha and Vazirani [1091], Vazirani [1217], and Chor and Goldreich [258].

§5.3

The idea of using a one-way function f for generating pseudorandom bit sequences is due to Shamir [1112]. Shamir illustrated why it is difficult to prove that such ad-hoc generators are cryptographically secure without imposing some further assumptions on f . Algorithm 5.11 is from Appendix C of the ANSI X9.17 standard [37]; it is one of the approved methods for pseudorandom bit generation listed in FIPS 186 [406]. Meyer and Matyas [859, pp.316-317] describe another DES-based pseudorandom bit generator whose output is intended for use as data-encrypting keys. The four algorithms of §5.3.2 for generating DSA parameters are from FIPS 186.

§5.4

Standard references on statistics include Hogg and Tanis [559] and Wackerly, Mendenhall, and Scheaffer [1226]. Tables 5.1 and 5.2 were generated using the Maple symbolic algebra system [240]. Golomb's randomness postulates (§5.4.3) were proposed by Golomb [498].

The five statistical tests for local randomness outlined in §5.4.4 are from Beker and Piper [84]. The serial test (§5.4.4(ii)) is due to Good [508]. It was generalized to subsequences of length greater than 2 by Marsaglia [782] who called it the *overlapping m -tuple test*, and later by Kimberley [674] who called it the *generalized serial test*. The underlying distribution theories of the serial test and the runs test (§5.4.4(iv)) were analyzed by Good [507] and Mood [897], respectively. Gustafson [531] considered alternative statistics for the runs test and the autocorrelation test (§5.4.4(v)).

There are numerous other statistical tests of local randomness. Many of these tests, including the gap test, coupon collector's test, permutation test, run test, maximum-of- t test, collision test, serial test, correlation test, and spectral test are described by Knuth [692]. The poker test as formulated by Knuth [692, p.62] is quite different from that of §5.4.4(iii). In the former, a sample sequence is divided into m -bit blocks, each of which is further subdivided into l -bit sub-blocks (for some divisor l of m). The number of m -bit blocks having r distinct l -bit sub-blocks ($1 \leq r \leq m/l$) is counted and compared to the corresponding expected numbers for random sequences. Erdmann [372] gives a detailed exposition of many

of these tests, and applies them to sample output sequences of six pseudorandom bit generators. Gustafson et al. [533] describe a computer package which implements various statistical tests for assessing the strength of a pseudorandom bit generator. Gustafson, Dawson, and Golić [532] proposed a new *repetition test* which measures the number of repetitions of l -bit blocks. The test requires a count of the number of patterns repeated, but does not require the frequency of each pattern. For this reason, it is feasible to apply this test for larger values of l (e.g. $l = 64$) than would be permissible by the poker test or Maurer's universal statistical test (Algorithm 5.33). Two spectral tests have been developed, one based on the discrete Fourier transform by Gait [437], and one based on the Walsh transform by Yuen [1260]. For extensions of these spectral tests, see Erdmann [372] and Feldman [389].

FIPS 140-1 [401] specifies security requirements for the design and implementation of cryptographic modules, including random and pseudorandom bit generators, for protecting (U.S. government) unclassified information.

The universal statistical test (Algorithm 5.33) is due to Maurer [813] and was motivated by source coding algorithms of Elias [371] and Willems [1245]. The class of defects that the test is able to detect consists of those that can be modeled by an ergodic stationary source with limited memory; Maurer argues that this class includes the possible defects that could occur in a practical implementation of a random bit generator. Table 5.3 is due to Maurer [813], who provides derivations of formulae for the mean and variance of the statistic X_u .

§5.5

Blum and Micali [166] presented the following general construction for CSPRBGs. Let D be a finite set, and let $f: D \rightarrow D$ be a permutation that can be efficiently computed. Let $B: D \rightarrow \{0, 1\}$ be a Boolean predicate with the property that $B(x)$ is hard to compute given only $x \in D$, however, $B(x)$ can be efficiently computed given $y = f^{-1}(x)$. The output sequence z_1, z_2, \dots, z_l corresponding to a seed $x_0 \in D$ is obtained by computing $x_i = f(x_{i-1})$, $z_i = B(x_i)$, for $1 \leq i \leq l$. This generator can be shown to pass the next-bit test (Definition 5.6). Blum and Micali [166] proposed the first concrete instance of a CSPRBG, called the *Blum-Micali generator*. Using the notation introduced above, their method can be described as follows. Let p be a large prime, and α a generator of \mathbb{Z}_p^* . Define $D = \mathbb{Z}_p^* = \{1, 2, \dots, p-1\}$. The function $f: D \rightarrow D$ is defined by $f(x) = \alpha^x \bmod p$. The function $B: D \rightarrow \{0, 1\}$ is defined by $B(x) = 1$ if $0 \leq \log_\alpha x \leq (p-1)/2$, and $B(x) = 0$ if $\log_\alpha x > (p-1)/2$. Assuming the intractability of the discrete logarithm problem in \mathbb{Z}_p^* (§3.6; see also §3.9.1), the Blum-Micali generator was proven to satisfy the next-bit test. Long and Wigderson [772] improved the efficiency of the Blum-Micali generator by simultaneously extracting $O(\lg \lg p)$ bits (cf. §3.9.1) from each x_i . Kaliski [650, 651] modified the Blum-Micali generator so that the security depends on the discrete logarithm problem in the group of points on an elliptic curve defined over a finite field.

The RSA pseudorandom bit generator (Algorithm 5.35) and the improvement mentioned in Note 5.36 are due to Alexi et al. [23]. The Micali-Schnorr improvement of the RSA PRBG (Algorithm 5.37) is due to Micali and Schnorr [867], who also described a method that transforms any CSPRBG into one that can be accelerated by parallel evaluation. The method of parallelization is *perfect*: m parallel processors speed the generation of pseudorandom bits by a factor of m .

Algorithm 5.40 is due to Blum, Blum, and Shub [160], who showed that their pseudorandom bit generator is cryptographically secure assuming the intractability of the quadratic residuosity problem (§3.4). Vazirani and Vazirani [1218] established a stronger result regarding the security of this generator by proving it cryptographically secure under the weaker assumption that integer factorization is intractable. The improvement mentioned in

Note 5.41 is due to Vazirani and Vazirani. Alexi et al. [23] proved analogous results for the *modified-Rabin generator*, which differs as follows from the Blum-Blum-Shub generator: in step 3.1 of Algorithm 5.40, let $\bar{x} = x_{i-1}^2 \bmod n$; if $\bar{x} < n/2$, then $x_i = \bar{x}$; otherwise, $x_i = n - \bar{x}$.

Impagliazzo and Naor [569] devised efficient constructions for a CSPRBG and for a universal one-way hash function which are provably as secure as the subset sum problem. Fischer and Stern [411] presented a simple and efficient CSPRBG which is provably as secure as the *syndrome decoding problem*.

Yao [1258] showed how to obtain a CSPRBG using any one-way permutation. Levin [761] generalized this result and showed how to obtain a CSPRBG using any one-way function. For further refinements, see Goldreich, Krawczyk, and Luby [470], Impagliazzo, Levin, and Luby [568], and Håstad [545].

A *random function* $f: \{0, 1\}^n \rightarrow \{0, 1\}^n$ is a function which assigns independent and random values $f(x) \in \{0, 1\}^n$ to all arguments $x \in \{0, 1\}^n$. Goldreich, Goldwasser, and Micali [468] introduced a computational complexity measure of the randomness of functions. They defined a function to be *poly-random* if no polynomial-time algorithm can distinguish between values of the function and true random strings, even when the algorithm is permitted to select the arguments to the function. Goldreich, Goldwasser, and Micali presented an algorithm for constructing poly-random functions assuming the existence of one-way functions. This theory was applied by Goldreich, Goldwasser, and Micali [467] to develop provably secure protocols for the (essentially) storageless distribution of secret identification numbers, message authentication with timestamping, dynamic hashing, and identify friend or foe systems. Luby and Rackoff [776] showed how poly-random permutations can be efficiently constructed from poly-random functions. This result was used, together with some of the design principles of DES, to show how any CSPRBG can be used to construct a symmetric-key block cipher which is provably secure against chosen-plaintext attack. A simplified and generalized treatment of Luby and Rackoff's construction was given by Maurer [816].

Schnorr [1096] used Luby and Rackoff's poly-random permutation generator to construct a pseudorandom bit generator that was claimed to pass all statistical tests depending only on a small fraction of the output sequence, even when infinite computational resources are available. Rueppel [1079] showed that this claim is erroneous, and demonstrated that the generator can be distinguished from a truly random bit generator using only a small number of output bits. Maurer and Massey [821] extended Schnorr's work, and proved the existence of pseudorandom bit generators that pass all statistical tests depending only on a small fraction of the output sequence, even when infinite computational resources are available. The security of the generators does not rely on any unproved hypothesis, but rather on the assumption that the adversary can access only a limited number of bits of the generated sequence. This work is primarily of theoretical interest since no such polynomial-time generators are known.

Chapter 7

Block Ciphers

Contents in Brief

7.1	Introduction and overview	223
7.2	Background and general concepts	224
7.3	Classical ciphers and historical development	237
7.4	DES	250
7.5	FEAL	259
7.6	IDEA	263
7.7	SAFER, RC5, and other block ciphers	266
7.8	Notes and further references	271

7.1 Introduction and overview

Symmetric-key block ciphers are the most prominent and important elements in many cryptographic systems. Individually, they provide confidentiality. As a fundamental building block, their versatility allows construction of pseudorandom number generators, stream ciphers, MACs, and hash functions. They may furthermore serve as a central component in message authentication techniques, data integrity mechanisms, entity authentication protocols, and (symmetric-key) digital signature schemes. This chapter examines symmetric-key block ciphers, including both general concepts and details of specific algorithms. Public-key block ciphers are discussed in Chapter 8.

No block cipher is ideally suited for all applications, even one offering a high level of security. This is a result of inevitable tradeoffs required in practical applications, including those arising from, for example, speed requirements and memory limitations (e.g., code size, data size, cache memory), constraints imposed by implementation platforms (e.g., hardware, software, chipcards), and differing tolerances of applications to properties of various modes of operation. In addition, efficiency must typically be traded off against security. Thus it is beneficial to have a number of candidate ciphers from which to draw.

Of the many block ciphers currently available, focus in this chapter is given to a subset of high profile and/or well-studied algorithms. While not guaranteed to be more secure than other published candidate ciphers (indeed, this status changes as new attacks become known), emphasis is given to those of greatest practical interest. Among these, DES is paramount; FEAL has received both serious commercial backing and a large amount of independent cryptographic analysis; and IDEA (originally proposed as a DES replacement) is widely known and highly regarded. Other recently proposed ciphers of both high promise and high profile (in part due to the reputation of their designers) are SAFER and RC5. Additional ciphers are presented in less detail.

(i) Mono-alphabetic substitution

Suppose the ciphertext and plaintext character sets are the same. Let $m = m_1m_2m_3 \dots$ be a plaintext message consisting of juxtaposed characters $m_i \in \mathcal{A}$, where \mathcal{A} is some fixed character alphabet such as $\mathcal{A} = \{A, B, \dots, Z\}$. A *simple substitution cipher* or *mono-alphabetic substitution cipher* employs a permutation e over \mathcal{A} , with encryption mapping $E_e(m) = e(m_1)e(m_2)e(m_3) \dots$. Here juxtaposition indicates concatenation (rather than multiplication), and $e(m_i)$ is the character to which m_i is mapped by e . This corresponds to Definition 1.27.

7.48 Example (*trivial shift cipher/Caesar cipher*) A *shift cipher* is a simple substitution cipher with the permutation e constrained to an alphabetic shift through k characters for some fixed k . More precisely, if $|\mathcal{A}| = s$, and m_i is associated with the integer value i , $0 \leq i \leq s - 1$, then $c_i = e(m_i) = m_i + k \bmod s$. The decryption mapping is defined by $d(c_i) = c_i - k \bmod s$. For English text, $s = 26$, and characters A through Z are associated with integers 0 through 25. For $k = 1$, the message $m = \text{HAL}$ is encrypted to $c = \text{IBM}$. According to folklore, Julius Caesar used the key $k = 3$. \square

The shift cipher can be trivially broken because there are only $s = |\mathcal{A}|$ keys (e.g., $s = 26$) to exhaustively search. A similar comment holds for affine ciphers (Example 7.49). More generally, see Fact 7.68.

7.49 Example (*affine cipher – historical*) The affine cipher on a 26-letter alphabet is defined by $e_K(x) = ax + b \bmod 26$, where $0 \leq a, b \leq 25$. The key is (a, b) . Ciphertext $c = e_K(x)$ is decrypted using $d_K(c) = (c - b)a^{-1} \bmod 26$, with the necessary and sufficient condition for invertibility that $\gcd(a, 26) = 1$. Shift ciphers are a subclass defined by $a = 1$. \square

7.50 Note (*recognizing simple substitution*) Mono-alphabetic substitution alters the frequency of individual plaintext characters, but does not alter the frequency distribution of the overall character set. Thus, comparing ciphertext character frequencies to a table of expected letter frequencies (unigram statistics) in the plaintext language allows associations between ciphertext and plaintext characters. (E.g., if the most frequent plaintext character X occurred twelve times, then the ciphertext character that X maps to will occur twelve times).

(ii) Polygram substitution

A simple substitution cipher substitutes for single plaintext letters. In contrast, *polygram substitution ciphers* involve groups of characters being substituted by other groups of characters. For example, sequences of two plaintext characters (*digrams*) may be replaced by other digrams. The same may be done with sequences of three plaintext characters (*trigrams*), or more generally using n -grams.

In full digram substitution over an alphabet of 26 characters, the key may be any of the 26^2 digrams, arranged in a table with row and column indices corresponding to the first and second characters in the digram, and the table entries being the ciphertext digrams substituted for the plaintext pairs. There are then $(26^2)!$ keys.

7.51 Example (*Playfair cipher – historical*) A digram substitution may be defined by arranging the characters of a 25-letter alphabet (I and J are equated) in a 5×5 matrix M . Adjacent plaintext characters are paired. The pair (p_1, p_2) is replaced by the digram (c_3, c_4) as follows. If p_1 and p_2 are in distinct rows and columns, they define the corners of a submatrix (possibly M itself), with the remaining corners c_3 and c_4 ; c_3 is defined as the character in the same column as p_1 . If p_1 and p_2 are in a common row, c_3 is defined as the character immediately to the right of p_1 and c_4 that immediately right of p_2 (the first column is

7.56 Example (*single mixed alphabet Vigenère*) A simple substitution mapping defined by a general permutation e (not restricted to an alphabetic shift), followed by a simple Vigenère, is defined by the mapping $c_i = e(m_i) + k_i \bmod s$, with inverse $m_i = e^{-1}(c_i - k_i) \bmod s$. An alternative is a simple Vigenère followed by a simple substitution: $c_i = e(m_i + k_i \bmod s)$, with inverse $m_i = e^{-1}(c_i) - k_i \bmod s$. \square

7.57 Example (*full Vigenère*) In a simple Vigenère of period t , replace the mapping defined by the shift value k_i (for shifting character m_i) by a general permutation e_i of the alphabet. The result is the substitution mapping $c_i = e_i(m_i)$, where the subscript i in e_i is taken modulo t . The key consists of t permutations e_1, \dots, e_t . \square

7.58 Example (*running-key Vigenère*) If the keystream k_i of a simple Vigenère is as long as the plaintext, the cipher is called a *running-key cipher*. For example, the key may be meaningful text from a book. \square

While running-key ciphers prevent cryptanalysis by the Kasiski method (§7.3.5), if the key has redundancy, cryptanalysis exploiting statistical imbalances may nonetheless succeed. For example, when encrypting plaintext English characters using a meaningful text as a running key, cryptanalysis is possible based on the observation that a significant proportion of ciphertext characters results from the encryption of high-frequency running text characters with high-frequency plaintext characters.

7.59 Fact A running-key cipher can be strengthened by successively enciphering plaintext under two or more distinct running keys. For typical English plaintext and running keys, it can be shown that iterating four such encipherments appears unbreakable.

7.60 Definition An *auto-key cipher* is a cipher wherein the plaintext itself serves as the key (typically subsequent to the use of an initial priming key).

7.61 Example (*auto-key Vigenère*) In a running-key Vigenère (Example 7.58) with an s -character alphabet, define a *priming key* $k = k_1 k_2 \dots k_t$. Plaintext characters m_i are encrypted as $c_i = m_i + k_i \bmod s$ for $1 \leq i \leq t$ (simplest case: $t = 1$). For $i > t$, $c_i = (m_i + m_{i-t}) \bmod s$. An alternative involving more keying material is to replace the simple shift by a full Vigenère with permutations e_i , $1 \leq i \leq s$, defined by the key k_i or character m_i : for $1 \leq i \leq t$, $c_i = e_{k_i}(m_i)$, and for $i > t$, $c_i = e_{m_{i-t}}(m_i)$. \square

An alternative to Example 7.61 is to auto-key a cipher using the resulting ciphertext as the key: for example, for $i > t$, $c_i = (m_i + c_{i-t}) \bmod s$. This, however, is far less desirable, as it provides an eavesdropping cryptanalyst the key itself.

7.62 Example (*Vernam viewed as a Vigenère*) Consider a simple Vigenère defined by $c_i = m_i + k_i \bmod s$. If the keystream is truly random and independent – as long as the plaintext and never repeated (cf. Example 7.58) – this yields the unconditionally secure Vernam cipher (Definition 1.39; §6.1.1), generalized from a binary to an arbitrary alphabet. \square

7.3.4 Polyalphabetic cipher machines and rotors (historical)

The *Jefferson cylinder* is a deceptively simple device which implements a polyalphabetic substitution cipher; conceived in the late 18th century, it had remarkable cryptographic

strength for its time. Polyalphabetic substitution ciphers implemented by a class of rotor-based machines were the dominant cryptographic tool in World War II. Such machines, including the Enigma machine and those of Hagelin, have an alphabet which changes continuously for a very long period before repeating; this provides protection against Kasiski analysis and methods based on the index of coincidence (§7.3.5).

(i) Jefferson cylinder

The *Jefferson cylinder* (Figure 7.3) implements a polyalphabetic substitution cipher while avoiding complex machinery, extensive user computations, and Vigenère tableaux. A solid cylinder 6 inches long is sliced into 36 disks. A rod inserted through the cylinder axis allows the disks to rotate. The periphery of each disk is divided into 26 parts. On each disk, the letters A–Z are inscribed in a (different) random ordering. Plaintext messages are encrypted in 36-character blocks. A reference bar is placed along the cylinder’s length. Each of the 36 wheels is individually rotated to bring the appropriate character (matching the plaintext block) into position along the reference line. The 25 other parallel reference positions then each define a ciphertext, from which (in an early instance of randomized encryption) one is selected as the ciphertext to transmit.

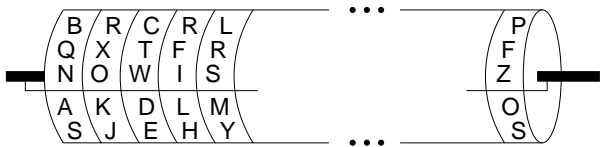


Figure 7.3: The Jefferson cylinder.

The second party possesses a cylinder with identically marked and ordered disks (1–36). The ciphertext is decrypted by rotating each of the 36 disks to obtain characters along a fixed reference line matching the ciphertext. The other 25 reference positions are examined for a recognizable plaintext. If the original message is not recognizable (e.g., random data), both parties agree beforehand on an index 1 through 25 specifying the offset between plaintext and ciphertext lines.

To accommodate plaintext digits 0–9 without extra disk sections, each digit is permanently assigned to one of 10 letters (a,e,i,o,u,y and f,l,r,s) which is encrypted as above but annotated with an overhead dot, identifying that the procedure must be reversed. Re-ordering disks (1 through 36) alters the polyalphabetic substitution key. The number of possible orderings is $36! \approx 3.72 \times 10^{41}$. Changing the ordering of letters on each disk affords $25!$ further mappings (per disk), but is more difficult in practice.

(ii) Rotor-based machines – technical overview

A simplified generic rotor machine (Figure 7.4) consists of a number of *rotors* (wired *code-wheels*) each implementing a different fixed mono-alphabetic substitution, mapping a character at its input face to one on its output face. A plaintext character input to the first rotor generates an output which is input to the second rotor, and so on, until the final ciphertext character emerges from the last. For fixed rotor positions, the bank of rotors collectively implements a mono-alphabetic substitution which is the composition of the substitutions defined by the individual rotors.

To provide polyalphabetic substitution, the encipherment of each plaintext character causes various rotors to move. The simplest case is an odometer-like movement, with a single rotor stepped until it completes a full revolution, at which time it steps the adjacent

Chapter 9

Hash Functions and Data Integrity

Contents in Brief

9.1	Introduction	321
9.2	Classification and framework	322
9.3	Basic constructions and general results	332
9.4	Unkeyed hash functions (MDCs)	338
9.5	Keyed hash functions (MACs)	352
9.6	Data integrity and message authentication	359
9.7	Advanced attacks on hash functions	368
9.8	Notes and further references	376

9.1 Introduction

Cryptographic hash functions play a fundamental role in modern cryptography. While related to conventional hash functions commonly used in non-cryptographic computer applications – in both cases, larger domains are mapped to smaller ranges – they differ in several important aspects. Our focus is restricted to cryptographic hash functions (hereafter, simply hash functions), and in particular to their use for data integrity and message authentication.

Hash functions take a message as input and produce an output referred to as a *hash-code*, *hash-result*, *hash-value*, or simply *hash*. More precisely, a hash function h maps bit-strings of arbitrary finite length to strings of fixed length, say n bits. For a domain D and range R with $h : D \rightarrow R$ and $|D| > |R|$, the function is many-to-one, implying that the existence of *collisions* (pairs of inputs with identical output) is unavoidable. Indeed, restricting h to a domain of t -bit inputs ($t > n$), if h were “random” in the sense that all outputs were essentially equiprobable, then about 2^{t-n} inputs would map to each output, and two randomly chosen inputs would yield the same output with probability 2^{-n} (independent of t). The basic idea of cryptographic hash functions is that a hash-value serves as a compact representative image (sometimes called an *imprint*, *digital fingerprint*, or *message digest*) of an input string, and can be used as if it were uniquely identifiable with that string.

Hash functions are used for data integrity in conjunction with digital signature schemes, where for several reasons a message is typically hashed first, and then the hash-value, as a representative of the message, is signed in place of the original message (see Chapter 11). A distinct class of hash functions, called message authentication codes (MACs), allows message authentication by symmetric techniques. MAC algorithms may be viewed as hash functions which take two functionally distinct inputs, a message and a secret key, and produce a fixed-size (say n -bit) output, with the design intent that it be infeasible in

9.15 Example (*one-wayness w.r.t. two inputs*) Consider $f(x, k) = E_k(x)$, where E represents DES. This is not a one-way function of the joint input (x, k) , because given any function value $y = f(x, k)$, one can choose any key k' and compute $x' = E_{k'}^{-1}(y)$ yielding a preimage (x', k') . Similarly, $f(x, k)$ is not a one-way function of x if k is known, as given $y = f(x, k)$ and k , decryption of y using k yields x . (However, a “black-box” which computes $f(x, k)$ for fixed, externally-unknown k is a one-way function of x .) In contrast, $f(x, k)$ is a one-way function of k ; given $y = f(x, k)$ and x , it is not known how to find a preimage k in less than about 2^{55} operations. (This latter concept is utilized in one-time digital signature schemes – see §11.6.2.) \square

9.16 Example (*OWF - multiplication of large primes*) For appropriate choices of primes p and q , $f(p, q) = pq$ is a one-way function: given p and q , computing $n = pq$ is easy, but given n , finding p and q , i.e., *integer factorization*, is difficult. RSA and many other cryptographic systems rely on this property (see Chapter 3, Chapter 8). Note that contrary to many one-way functions, this function f does not have properties resembling a “random” function. \square

9.17 Example (*OWF - exponentiation in finite fields*) For most choices of appropriately large primes p and any element $\alpha \in \mathbb{Z}_p^*$ of sufficiently large multiplicative order (e.g., a generator), $f(x) = \alpha^x \bmod p$ is a one-way function. (For example, p must not be such that all the prime divisors of $p - 1$ are small, otherwise the discrete log problem is feasible by the Pohlig-Hellman algorithm of §3.6.4.) $f(x)$ is easily computed given α , x , and p using the square-and-multiply technique (Algorithm 2.143), but for most choices p it is difficult, given (y, p, α) , to find an x in the range $0 \leq x \leq p - 2$ such that $\alpha^x \bmod p = y$, due to the apparent intractability of the discrete logarithm problem (§3.6). Of course, for specific values of $f(x)$ the function can be inverted trivially. For example, the respective preimages of 1 and -1 are known to be 0 and $(p - 1)/2$, and by computing $f(x)$ for any small set of values for x (e.g., $x = 1, 2, \dots, 10$), these are also known. However, for *essentially* all y in the range, the preimage of y is difficult to find. \square

9.2.5 Relationships between properties

In this section several relationships between the hash function properties stated in the preceding section are examined.

9.18 Fact Collision resistance implies 2nd-preimage resistance of hash functions.

Justification. Suppose h has collision resistance. Fix an input x_j . If h does not have 2nd-preimage resistance, then it is feasible to find a distinct input x_i such that $h(x_i) = h(x_j)$, in which case (x_i, x_j) is a pair of distinct inputs hashing to the same output, contradicting collision resistance.

9.19 Remark (*one-way vs. preimage and 2nd-preimage resistant*) While the term “one-way” is generally taken to mean preimage resistant, in the hash function literature it is sometimes also used to imply that a function is 2nd-preimage resistant or computationally non-invertible. (*Computationally non-invertible* is a more explicit term for preimage resistance when preimages are unique, e.g., for one-way permutations. In the case that two or more preimages exist, a function fails to be computationally non-invertible if any one can be found.) This causes ambiguity as 2nd-preimage resistance does not guarantee preimage-resistance (Note 9.20), nor does preimage resistance guarantee 2nd-preimage resistance (Example 9.11); see also Remark 9.10. An attempt is thus made to avoid unqualified use of the term “one-way”.

(ii) MD5

MD5 (Algorithm 9.51) was designed as a strengthened version of MD4, prior to actual MD4 collisions being found. It has enjoyed widespread use in practice. It has also now been found to have weaknesses (Remark 9.52).

The changes made to obtain MD5 from MD4 are as follows:

1. addition of a fourth round of 16 steps, and a Round 4 function
2. replacement of the Round 2 function by a new function
3. modification of the access order for message words in Rounds 2 and 3
4. modification of the shift amounts (such that shifts differ in distinct rounds)
5. use of unique additive constants in each of the 4×16 steps, based on the integer part of $2^{32} \cdot \sin(j)$ for step j (requiring overall, 256 bytes of storage)
6. addition of output from the previous step into each of the 64 steps.

9.51 Algorithm MD5 hash function

INPUT: bitstring x of arbitrary bitlength $b \geq 0$. (For notation, see Table 9.7.)

OUTPUT: 128-bit hash-code of x . (See Table 9.6 for test vectors.)

MD5 is obtained from MD4 by making the following changes.

1. *Notation.* Replace the Round 2 function by: $g(u, v, w) \stackrel{\text{def}}{=} uw \vee v\overline{w}$.
Define a Round 4 function: $k(u, v, w) \stackrel{\text{def}}{=} v \oplus (u \vee \overline{w})$.
2. *Definition of constants.* Redefine unique additive constants:
 $y[j] = \text{first 32 bits of binary value } \text{abs}(\sin(j+1))$, $0 \leq j \leq 63$, where j is in radians and “abs” denotes absolute value. Redefine access order for words in Rounds 2 and 3, and define for Round 4:
 $z[16..31] = [1, 6, 11, 0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12]$,
 $z[32..47] = [5, 8, 11, 14, 1, 4, 7, 10, 13, 0, 3, 6, 9, 12, 15, 2]$,
 $z[48..63] = [0, 7, 14, 5, 12, 3, 10, 1, 8, 15, 6, 13, 4, 11, 2, 9]$.
Redefine number of bit positions for left shifts (rotates):
 $s[0..15] = [7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22]$,
 $s[16..31] = [5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20]$,
 $s[32..47] = [4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23]$,
 $s[48..63] = [6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21]$.
3. *Preprocessing.* As in MD4.
4. *Processing.* In each of Rounds 1, 2, and 3, replace “ $B \leftarrow (t \leftarrow s[j])$ ” by “ $B \leftarrow B + (t \leftarrow s[j])$ ”. Also, immediately following Round 3 add:
(Round 4) For j from 48 to 63 do the following:
 $t \leftarrow (A + k(B, C, D) + X[z[j]] + y[j])$, $(A, B, C, D) \leftarrow (D, B + (t \leftarrow s[j]), B, C)$.
5. *Completion.* As in MD4.

9.52 Remark (*MD5 compression function collisions*) While no collisions for MD5 have yet been found (cf. Table 9.3), collisions have been found for the MD5 compression function. More specifically, these are called collisions for random IV. (See §9.7.2, and in particular Definition 9.97 and Note 9.98.)

(iii) SHA-1

The Secure Hash Algorithm (SHA-1), based on MD4, was proposed by the U.S. National Institute for Standards and Technology (NIST) for certain U.S. federal government applications. The main differences of SHA-1 from MD4 are as follows:

1. The hash-value is 160 bits, and five (vs. four) 32-bit chaining variables are used.
2. The compression function has four rounds instead of three, using the MD4 step functions f , g , and h as follows: f in the first, g in the third, and h in both the second and fourth rounds. Each round has 20 steps instead of 16.
3. Within the compression function, each 16-word message block is expanded to an 80-word block, by a process whereby each of the last 64 of the 80 words is the XOR of 4 words from earlier positions in the expanded block. These 80 words are then input one-word-per-step to the 80 steps.
4. The core step is modified as follows: the only rotate used is a constant 5-bit rotate; the fifth working variable is added into each step result; message words from the expanded message block are accessed sequentially; and C is updated as B rotated left 30 bits, rather than simply B .
5. SHA-1 uses four non-zero additive constants, whereas MD4 used three constants only two of which were non-zero.

The byte ordering used for converting between streams of bytes and 32-bit words in the official SHA-1 specification is big-endian (see Note 9.48); this differs from MD4 which is little-endian.

9.53 Algorithm Secure Hash Algorithm – revised (SHA-1)

INPUT: bitstring x of bitlength $b \geq 0$. (For notation, see Table 9.7.)

OUTPUT: 160-bit hash-code of x . (See Table 9.6 for test vectors.)

SHA-1 is defined (with reference to MD4) by making the following changes.

1. *Notation.* As in MD4.
2. *Definition of constants.* Define a fifth IV to match those in MD4: $h_5 = 0xc3d2e1f0$. Define per-round integer additive constants: $y_1 = 0x5a827999$, $y_2 = 0x6ed9eba1$, $y_3 = 0x8f1bbcdc$, $y_4 = 0xca62c1d6$. (No order for accessing source words, or specification of bit positions for left shifts is required.)
3. *Overall preprocessing.* Pad as in MD4, except the final two 32-bit words specifying the bitlength b is appended with most significant word preceding least significant. As in MD4, the formatted input is $16m$ 32-bit words: $x_0x_1 \dots x_{16m-1}$. Initialize chaining variables: $(H_1, H_2, H_3, H_4, H_5) \leftarrow (h_1, h_2, h_3, h_4, h_5)$.
4. *Processing.* For each i from 0 to $m - 1$, copy the i^{th} block of sixteen 32-bit words into temporary storage: $X[j] \leftarrow x_{16i+j}$, $0 \leq j \leq 15$, and process these as below in four 20-step rounds before updating the chaining variables:
 (expand 16-word block into 80-word block; let X_j denote $X[j]$)
 for j from 16 to 79, $X_j \leftarrow ((X_{j-3} \oplus X_{j-8} \oplus X_{j-14} \oplus X_{j-16}) \leftarrow 1)$.
 (initialize working variables) $(A, B, C, D, E) \leftarrow (H_1, H_2, H_3, H_4, H_5)$.
 (Round 1) For j from 0 to 19 do the following:
 $t \leftarrow ((A \leftarrow 5) + f(B, C, D) + E + X_j + y_1)$,
 $(A, B, C, D, E) \leftarrow (t, A, B \leftarrow 30, C, D)$.
 (Round 2) For j from 20 to 39 do the following:
 $t \leftarrow ((A \leftarrow 5) + h(B, C, D) + E + X_j + y_2)$,
 $(A, B, C, D, E) \leftarrow (t, A, B \leftarrow 30, C, D)$.

- (Round 3) For j from 40 to 59 do the following:
 $t \leftarrow ((A \leftarrow 5) + g(B, C, D) + E + X_j + y_3)$,
 $(A, B, C, D, E) \leftarrow (t, A, B \leftarrow 30, C, D)$.
(Round 4) For j from 60 to 79 do the following:
 $t \leftarrow ((A \leftarrow 5) + h(B, C, D) + E + X_j + y_4)$,
 $(A, B, C, D, E) \leftarrow (t, A, B \leftarrow 30, C, D)$.
(update chaining values)
 $(H_1, H_2, H_3, H_4, H_5) \leftarrow (H_1 + A, H_2 + B, H_3 + C, H_4 + D, H_5 + E)$.
5. *Completion.* The hash-value is: $H_1 || H_2 || H_3 || H_4 || H_5$
(with first and last bytes the high- and low-order bytes of H_1, H_5 , respectively).

9.54 Remark (*security of SHA-1*) Compared to 128-bit hash functions, the 160-bit hash-value of SHA-1 provides increased security against brute-force attacks. SHA-1 and RIPEMD-160 (see §9.4.2(iv)) presently appear to be of comparable strength; both are considered stronger than MD5 (Remark 9.52). In SHA-1, a significant effect of the expansion of 16-word message blocks to 80 words in the compression function is that any two distinct 16-word blocks yield 80-word values which differ in a larger number of bit positions, significantly expanding the number of bit differences among message words input to the compression function. The redundancy added by this preprocessing evidently adds strength.

(iv) RIPEMD-160

RIPEMD-160 (Algorithm 9.55) is a hash function based on MD4, taking into account knowledge gained in the analysis of MD4, MD5, and RIPEMD. The overall RIPEMD-160 compression function maps 21-word inputs (5-word chaining variable plus 16-word message block, with 32-bit words) to 5-word outputs. Each input block is processed in parallel by distinct versions (the *left line* and *right line*) of the compression function. The 160-bit outputs of the separate lines are combined to give a single 160-bit output.

Notation	Definition
$f(u, v, w)$	$u \oplus v \oplus w$
$g(u, v, w)$	$uv \vee \overline{u}w$
$h(u, v, w)$	$(u \vee \overline{v}) \oplus w$
$k(u, v, w)$	$uw \vee v\overline{w}$
$l(u, v, w)$	$u \oplus (v \vee \overline{w})$

Table 9.8: RIPEMD-160 round function definitions.

The RIPEMD-160 compression function differs from MD4 in the number of words of chaining variable, the number of rounds, the round functions themselves (Table 9.8), the order in which the input words are accessed, and the amounts by which results are rotated. The left and right computation lines differ from each other in these last two items, in their additive constants, and in the order in which the round functions are applied. This design is intended to improve resistance against known attack strategies. Each of the parallel lines uses the same IV as SHA-1. When writing the IV as a bitstring, little-endian ordering is used for RIPEMD-160 as in MD4 (vs. big-endian in SHA-1; see Note 9.48).

Chapter 14

Efficient Implementation

Contents in Brief

14.1	Introduction	591
14.2	Multiple-precision integer arithmetic	592
14.3	Multiple-precision modular arithmetic	599
14.4	Greatest common divisor algorithms	606
14.5	Chinese remainder theorem for integers	610
14.6	Exponentiation	613
14.7	Exponent recoding	627
14.8	Notes and further references	630

14.1 Introduction

Many public-key encryption and digital signature schemes, and some hash functions (see §9.4.3), require computations in \mathbb{Z}_m , the integers modulo m (m is a large positive integer which may or may not be a prime). For example, the RSA, Rabin, and ElGamal schemes require efficient methods for performing multiplication and exponentiation in \mathbb{Z}_m . Although \mathbb{Z}_m is prominent in many aspects of modern applied cryptography, other algebraic structures are also important. These include, but are not limited to, polynomial rings, finite fields, and finite cyclic groups. For example, the group formed by the points on an elliptic curve over a finite field has considerable appeal for various cryptographic applications. The efficiency of a particular cryptographic scheme based on any one of these algebraic structures will depend on a number of factors, such as parameter size, time-memory tradeoffs, processing power available, software and/or hardware optimization, and mathematical algorithms.

This chapter is concerned primarily with mathematical algorithms for efficiently carrying out computations in the underlying algebraic structure. Since many of the most widely implemented techniques rely on \mathbb{Z}_m , emphasis is placed on efficient algorithms for performing the basic arithmetic operations in this structure (addition, subtraction, multiplication, division, and exponentiation).

In some cases, several algorithms will be presented which perform the same operation. For example, a number of techniques for doing modular multiplication and exponentiation are discussed in §14.3 and §14.6, respectively. Efficiency can be measured in numerous ways; thus, it is difficult to definitively state which algorithm is the best. An algorithm may be efficient in the time it takes to perform a certain algebraic operation, but quite inefficient in the amount of storage it requires. One algorithm may require more code space than another. Depending on the environment in which computations are to be performed, one algorithm may be preferable over another. For example, current chipcard technology provides

very limited storage for both precomputed values and program code. For such applications, an algorithm which is less efficient in time but very efficient in memory requirements may be preferred.

The algorithms described in this chapter are those which, for the most part, have received considerable attention in the literature. Although some attempt is made to point out their relative merits, no detailed comparisons are given.

Chapter outline

§14.2 deals with the basic arithmetic operations of addition, subtraction, multiplication, squaring, and division for multiple-precision integers. §14.3 describes the basic arithmetic operations of addition, subtraction, and multiplication in \mathbb{Z}_m . Techniques described for performing modular reduction for an arbitrary modulus m are the classical method (§14.3.1), Montgomery's method (§14.3.2), and Barrett's method (§14.3.3). §14.3.4 describes a reduction procedure ideally suited to moduli of a special form. Greatest common divisor (gcd) algorithms are the topic of §14.4, including the binary gcd algorithm (§14.4.1) and Lehmer's gcd algorithm (§14.4.2). Efficient algorithms for performing extended gcd computations are given in §14.4.3. Modular inverses are also considered in §14.4.3. Garner's algorithm for implementing the Chinese remainder theorem can be found in §14.5. §14.6 is a treatment of several of the most practical exponentiation algorithms. §14.6.1 deals with exponentiation in general, without consideration of any special conditions. §14.6.2 looks at exponentiation when the base is variable and the exponent is fixed. §14.6.3 considers algorithms which take advantage of a fixed-base element and variable exponent. Techniques involving representing the exponent in non-binary form are given in §14.7; recoding the exponent may allow significant performance enhancements. §14.8 contains further notes and references.

14.2 Multiple-precision integer arithmetic

This section deals with the basic operations performed on multiple-precision integers: addition, subtraction, multiplication, squaring, and division. The algorithms presented in this section are commonly referred to as the *classical methods*.

14.2.1 Radix representation

Positive integers can be represented in various ways, the most common being *base 10*. For example, $a = 123$ base 10 means $a = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$. For machine computations, *base 2 (binary representation)* is preferable. If $a = 1111011$ base 2, then $a = 2^6 + 2^5 + 2^4 + 2^3 + 0 \cdot 2^2 + 2^1 + 2^0$.

14.1 Fact If $b \geq 2$ is an integer, then any positive integer a can be expressed uniquely as $a = a_n b^n + a_{n-1} b^{n-1} + \cdots + a_1 b + a_0$, where a_i is an integer with $0 \leq a_i < b$ for $0 \leq i \leq n$, and $a_n \neq 0$.

14.2 Definition The representation of a positive integer a as a sum of multiples of powers of b , as given in Fact 14.1, is called the *base b or radix b representation* of a .

14.3 Note (notation and terminology)

- (i) The base b representation of a positive integer a given in Fact 14.1 is usually written as $a = (a_n a_{n-1} \cdots a_1 a_0)_b$. The integers a_i , $0 \leq i \leq n$, are called *digits*. a_n is called the *most significant digit* or *high-order digit*; a_0 the *least significant digit* or *low-order digit*. If $b = 10$, the standard notation is $a = a_n a_{n-1} \cdots a_1 a_0$.
- (ii) It is sometimes convenient to pad high-order digits of a base b representation with 0's; such a padded number will also be referred to as the base b representation.
- (iii) If $(a_n a_{n-1} \cdots a_1 a_0)_b$ is the base b representation of a and $a_n \neq 0$, then the *precision* or *length* of a is $n+1$. If $n = 0$, then a is called a *single-precision integer*; otherwise, a is a *multiple-precision integer*. $a = 0$ is also a single-precision integer.

The division algorithm for integers (see Definition 2.82) provides an efficient method for determining the base b representation of a non-negative integer, for a given base b . This provides the basis for Algorithm 14.4.

14.4 Algorithm Radix b representation

INPUT: integers a and b , $a \geq 0$, $b \geq 2$.

OUTPUT: the base b representation $a = (a_n \cdots a_1 a_0)_b$, where $n \geq 0$ and $a_n \neq 0$ if $n \geq 1$.

1. $i \leftarrow 0$, $x \leftarrow a$, $q \leftarrow \lfloor \frac{x}{b} \rfloor$, $a_i \leftarrow x - qb$. ($\lfloor \cdot \rfloor$ is the floor function; see page 49.)
2. While $q > 0$, do the following:
 - 2.1 $i \leftarrow i + 1$, $x \leftarrow q$, $q \leftarrow \lfloor \frac{x}{b} \rfloor$, $a_i \leftarrow x - qb$.
3. Return $((a_i a_{i-1} \cdots a_1 a_0))$.

14.5 Fact If $(a_n a_{n-1} \cdots a_1 a_0)_b$ is the base b representation of a and k is a positive integer, then $(u_l u_{l-1} \cdots u_1 u_0)_{b^k}$ is the base b^k representation of a , where $l = \lceil (n+1)/k \rceil - 1$, $u_i = \sum_{j=0}^{k-1} a_{ik+j} b^j$ for $0 \leq i \leq l-1$, and $u_l = \sum_{j=0}^{n-lk} a_{lk+j} b^j$.

14.6 Example (radix b representation) The base 2 representation of $a = 123$ is $(1111011)_2$. The base 4 representation of a is easily obtained from its base 2 representation by grouping digits in pairs from the right: $a = ((1)_2(11)_2(10)_2(11)_2)_4 = (1323)_4$. \square

Representing negative numbers

Negative integers can be represented in several ways. Two commonly used methods are:

1. *signed-magnitude representation*
2. *complement representation*.

These methods are described below. The algorithms provided in this chapter all assume a signed-magnitude representation for integers, with the sign digit being implicit.

(i) Signed-magnitude representation

The *sign* of an integer (i.e., either positive or negative) and its *magnitude* (i.e., absolute value) are represented separately in a *signed-magnitude representation*. Typically, a positive integer is assigned a sign digit 0, while a negative integer is assigned a sign digit $b-1$. For n -digit radix b representations, only $2b^{n-1}$ sequences out of the b^n possible sequences are utilized: precisely $b^{n-1} - 1$ positive integers and $b^{n-1} - 1$ negative integers can be represented, and 0 has two representations. Table 14.1 illustrates the binary signed-magnitude representation of the integers in the range $[7, -7]$.

Signed-magnitude representation has the drawback that when certain operations (such as addition and subtraction) are performed, the sign digit must be checked to determine the appropriate manner to perform the computation. Conditional branching of this type can be costly when many operations are performed.

(ii) Complement representation

Addition and subtraction using *complement representation* do not require the checking of the sign digit. Non-negative integers in the range $[0, b^{n-1} - 1]$ are represented by base b sequences of length n with the high-order digit being 0. Suppose x is a positive integer in this range represented by the sequence $(x_n x_{n-1} \cdots x_1 x_0)_b$ where $x_n = 0$. Then $-x$ is represented by the sequence $\bar{x} = (\bar{x}_n \bar{x}_{n-1} \cdots \bar{x}_1 \bar{x}_0) + 1$ where $\bar{x}_i = b - 1 - x_i$ and $+$ is the standard addition with carry. Table 14.1 illustrates the binary complement representation of the integers in the range $[-7, 7]$. In the binary case, complement representation is referred to as *two's complement representation*.

Sequence	Signed-magnitude	Two's complement	Sequence	Signed-magnitude	Two's complement
0111	7	7	1111	-7	-1
0110	6	6	1110	-6	-2
0101	5	5	1101	-5	-3
0100	4	4	1100	-4	-4
0011	3	3	1011	-3	-5
0010	2	2	1010	-2	-6
0001	1	1	1001	-1	-7
0000	0	0	1000	-0	-8

Table 14.1: Signed-magnitude and two's complement representations of integers in $[-7, 7]$.

14.2.2 Addition and subtraction

Addition and subtraction are performed on two integers having the same number of base b digits. To add or subtract two integers of different lengths, the smaller of the two integers is first padded with 0's on the left (i.e., in the high-order positions).

14.7 Algorithm Multiple-precision addition

INPUT: positive integers x and y , each having $n + 1$ base b digits.

OUTPUT: the sum $x + y = (w_{n+1} w_n \cdots w_1 w_0)_b$ in radix b representation.

1. $c \leftarrow 0$ (c is the carry digit).
2. For i from 0 to n do the following:
 - 2.1 $w_i \leftarrow (x_i + y_i + c) \bmod b$.
 - 2.2 If $(x_i + y_i + c) < b$ then $c \leftarrow 0$; otherwise $c \leftarrow 1$.
3. $w_{n+1} \leftarrow c$.
4. Return($(w_{n+1} w_n \cdots w_1 w_0)$).

14.8 Note (*computational efficiency*) The base b should be chosen so that $(x_i + y_i + c) \bmod b$ can be computed by the hardware on the computing device. Some processors have instruction sets which provide an add-with-carry to facilitate multiple-precision addition.

14.9 Algorithm Multiple-precision subtraction

INPUT: positive integers x and y , each having $n + 1$ base b digits, with $x \geq y$.

OUTPUT: the difference $x - y = (w_n w_{n-1} \cdots w_1 w_0)_b$ in radix b representation.

1. $c \leftarrow 0$.
2. For i from 0 to n do the following:
 - 2.1 $w_i \leftarrow (x_i - y_i + c) \bmod b$.
 - 2.2 If $(x_i - y_i + c) \geq 0$ then $c \leftarrow 0$; otherwise $c \leftarrow -1$.
3. Return($(w_n w_{n-1} \cdots w_1 w_0)$).

14.10 Note (eliminating the requirement $x \geq y$) If the relative magnitudes of the integers x and y are unknown, then Algorithm 14.9 can be modified as follows. On termination of the algorithm, if $c = -1$, then repeat Algorithm 14.9 with $x = (00 \cdots 00)_b$ and $y = (w_n w_{n-1} \cdots w_1 w_0)_b$. Conditional checking on the relative magnitudes of x and y can also be avoided by using a complement representation (§14.2.1(ii)).

14.11 Example (modified subtraction) Let $x = 3996879$ and $y = 4637923$ in base 10, so that $x < y$. Table 14.2 shows the steps of the modified subtraction algorithm (cf. Note 14.10). \square

First execution of Algorithm 14.9							
i	6	5	4	3	2	1	0
x_i	3	9	9	6	8	7	9
y_i	4	6	3	7	9	2	3
w_i	9	3	5	8	9	5	6
c	-1	0	0	-1	-1	0	0

Second execution of Algorithm 14.9							
i	6	5	4	3	2	1	0
x_i	0	0	0	0	0	0	0
y_i	9	3	5	8	9	5	6
w_i	0	6	4	1	0	4	4
c	-1	-1	-1	-1	-1	-1	-1

Table 14.2: Modified subtraction (see Example 14.11).

14.2.3 Multiplication

Let x and y be integers expressed in radix b representation: $x = (x_n x_{n-1} \cdots x_1 x_0)_b$ and $y = (y_t y_{t-1} \cdots y_1 y_0)_b$. The product $x \cdot y$ will have at most $(n + t + 2)$ base b digits. Algorithm 14.12 is a reorganization of the standard pencil-and-paper method taught in grade school. A *single-precision* multiplication means the multiplication of two base b digits. If x_j and y_i are two base b digits, then $x_j \cdot y_i$ can be written as $x_j \cdot y_i = (uv)_b$, where u and v are base b digits, and u may be 0.

14.12 Algorithm Multiple-precision multiplication

INPUT: positive integers x and y having $n + 1$ and $t + 1$ base b digits, respectively.

OUTPUT: the product $x \cdot y = (w_{n+t+1} \cdots w_1 w_0)_b$ in radix b representation.

1. For i from 0 to $(n + t + 1)$ do: $w_i \leftarrow 0$.
2. For i from 0 to t do the following:
 - 2.1 $c \leftarrow 0$.
 - 2.2 For j from 0 to n do the following:

Compute $(uv)_b = w_{i+j} + x_j \cdot y_i + c$, and set $w_{i+j} \leftarrow v$, $c \leftarrow u$.
 - 2.3 $w_{i+n+1} \leftarrow c$.
3. Return($(w_{n+t+1} \cdots w_1 w_0)$).

14.16 Algorithm Multiple-precision squaringINPUT: positive integer $x = (x_{t-1}x_{t-2} \cdots x_1x_0)_b$.OUTPUT: $x \cdot x = x^2$ in radix b representation.

1. For i from 0 to $(2t - 1)$ do: $w_i \leftarrow 0$.
2. For i from 0 to $(t - 1)$ do the following:
 - 2.1 $(uv)_b \leftarrow w_{2i} + x_i \cdot x_i$, $w_{2i} \leftarrow v$, $c \leftarrow u$.
 - 2.2 For j from $(i + 1)$ to $(t - 1)$ do the following:
 $(uv)_b \leftarrow w_{i+j} + 2x_j \cdot x_i + c$, $w_{i+j} \leftarrow v$, $c \leftarrow u$.
 - 2.3 $w_{i+t} \leftarrow u$.
3. $(w) \leftarrow w_{2t-2} + x_{t-1} \cdot x_{t-1}$, $w_{2t-2} \leftarrow v$, $w_{2t-1} \leftarrow u$.
4. Return $((w_{2t-1}w_{2t-2} \cdots w_1w_0)_b)$.

14.17 Note (computational efficiency of Algorithm 14.16)

- (i) (*overflow*) In step 2.2, u can be larger than a single-precision integer. Since w_{i+j} is always set to v , $w_{i+j} \leq b - 1$. If $c \leq 2(b - 1)$, then $w_{i+j} + 2x_jx_i + c \leq (b - 1) + 2(b - 1)^2 + 2(b - 1) = (b - 1)(2b + 1)$, implying $0 \leq u \leq 2(b - 1)$. This value of u may exceed single-precision, and must be accommodated.
- (ii) (*number of operations*) The computationally intensive part of the algorithm is step 2. The number of single-precision multiplications is about $(t^2 + t)/2$, discounting the multiplication by 2. This is approximately one half of the single-precision multiplications required by Algorithm 14.12 (cf. Note 14.15(ii)).

14.18 Note (*squaring vs. multiplication in general*) Squaring a positive integer x (i.e., computing x^2) can at best be no more than twice as fast as multiplying distinct integers x and y . To see this, consider the identity $xy = ((x + y)^2 - (x - y)^2)/4$. Hence, $x \cdot y$ can be computed with two squarings (i.e., $(x + y)^2$ and $(x - y)^2$). Of course, a speed-up by a factor of 2 can be significant in many applications.

14.19 Example (*squaring*) Table 14.4 shows the steps performed by Algorithm 14.16 in squaring $x = 989$. Here, $t = 3$ and $b = 10$. □

i	j	$w_{2i} + x_i^2$	$w_{i+j} + 2x_jx_i + c$	u	v	w_5	w_4	w_3	w_2	w_1	w_0
0	—	0 + 81	—	8	1	0	0	0	0	0	1
	1	—	0 + 2 · 8 · 9 + 8	15	2	0	0	0	0	2	1
	2	—	0 + 2 · 9 · 9 + 15	17	7	0	0	0	7	2	1
				17	7	0	0	17	7	2	1
1	—	7 + 64	—	7	1	0	0	17	1	2	1
	2	—	17 + 2 · 9 · 8 + 7	16	8	0	0	8	1	2	1
				16	8	0	16	8	1	2	1
2	—	16 + 81	—	9	7	0	7	8	1	2	1
				9	7	9	7	8	1	2	1

Table 14.4: Multiple-precision squaring (see Example 14.19).

14.2.5 Division

Division is the most complicated and costly of the basic multiple-precision operations. Algorithm 14.20 computes the quotient q and remainder r in radix b representation when x is divided by y .

14.20 Algorithm Multiple-precision division

INPUT: positive integers $x = (x_n \cdots x_1 x_0)_b$, $y = (y_t \cdots y_1 y_0)_b$ with $n \geq t \geq 1$, $y_t \neq 0$.
 OUTPUT: the quotient $q = (q_{n-t} \cdots q_1 q_0)_b$ and remainder $r = (r_t \cdots r_1 r_0)_b$ such that $x = qy + r$, $0 \leq r < y$.

1. For j from 0 to $(n - t)$ do: $q_j \leftarrow 0$.
2. While $(x \geq yb^{n-t})$ do the following: $q_{n-t} \leftarrow q_{n-t} + 1$, $x \leftarrow x - yb^{n-t}$.
3. For i from n down to $(t + 1)$ do the following:
 - 3.1 If $x_i = y_t$ then set $q_{i-t-1} \leftarrow b - 1$; otherwise set $q_{i-t-1} \leftarrow \lfloor (x_i b + x_{i-1}) / y_t \rfloor$.
 - 3.2 While $(q_{i-t-1}(y_t b + y_{t-1}) > x_i b^2 + x_{i-1} b + x_{i-2})$ do: $q_{i-t-1} \leftarrow q_{i-t-1} - 1$.
 - 3.3 $x \leftarrow x - q_{i-t-1} y b^{i-t-1}$.
 - 3.4 If $x < 0$ then set $x \leftarrow x + y b^{i-t-1}$ and $q_{i-t-1} \leftarrow q_{i-t-1} - 1$.
4. $r \leftarrow x$.
5. Return(q, r).

14.21 Example (multiple-precision division) Let $x = 721948327$, $y = 84461$, so that $n = 8$ and $t = 4$. Table 14.5 illustrates the steps in Algorithm 14.20. The last row gives the quotient $q = 8547$ and the remainder $r = 60160$. \square

i	q_4	q_3	q_2	q_1	q_0	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
—	0	0	0	0	0	7	2	1	9	4	8	3	2	7
8	0	9	0	0	0	7	2	1	9	4	8	3	2	7
		8	0	0	0			4	6	2	6	0	3	2
7		8	5	0	0			4	0	2	9	8	2	7
6		8	5	5	0			4	0	2	9	8	2	7
			8	5	4	0			6	5	1	3	8	7
5			8	5	4	8			6	5	1	3	8	7
			8	5	4	7				6	0	1	6	0

Table 14.5: Multiple-precision division (see Example 14.21).

14.22 Note (comments on Algorithm 14.20)

- (i) Step 2 of Algorithm 14.20 is performed at most once if $y_t \geq \lfloor \frac{b}{2} \rfloor$ and b is even.
- (ii) The condition $n \geq t \geq 1$ can be replaced by $n \geq t \geq 0$, provided one takes $x_j = y_j = 0$ whenever a subscript $j < 0$ is encountered in the algorithm.

14.23 Note (normalization) The estimate for the quotient digit q_{i-t-1} in step 3.1 of Algorithm 14.20 is never less than the true value of the quotient digit. Furthermore, if $y_t \geq \lfloor \frac{b}{2} \rfloor$, then step 3.2 is repeated no more than twice. If step 3.1 is modified so that $q_{i-t-1} \leftarrow \lfloor (x_i b^2 + x_{i-1} b + x_{i-2}) / (y_t b + y_{t-1}) \rfloor$, then the estimate is almost always correct and step 3.2 is

never repeated more than once. One can always guarantee that $y_t \geq \lfloor \frac{b}{2} \rfloor$ by replacing the integers x, y by $\lambda x, \lambda y$ for some suitable choice of λ . The quotient of λx divided by λy is the same as that of x by y ; the remainder is λ times the remainder of x divided by y . If the base b is a power of 2 (as in many applications), then the choice of λ should be a power of 2; multiplication by λ is achieved by simply left-shifting the binary representations of x and y . Multiplying by a suitable choice of λ to ensure that $y_t \geq \lfloor \frac{b}{2} \rfloor$ is called *normalization*. Example 14.24 illustrates the procedure.

14.24 Example (normalized division) Take $x = 73418$ and $y = 267$. Normalize x and y by multiplying each by $\lambda = 3$: $x' = 3x = 220254$ and $y' = 3y = 801$. Table 14.6 shows the steps of Algorithm 14.20 as applied to x' and y' . When x' is divided by y' , the quotient is 274, and the remainder is 780. When x is divided by y , the quotient is also 274 and the remainder is $780/3 = 260$. \square

i	q_3	q_2	q_1	q_0	x_5	x_4	x_3	x_2	x_1	x_0
—	0	0	0	0	2	2	0	2	5	4
5	0	2	0	0		6	0	0	5	4
4		2	7	0			3	9	8	4
3		2	7	4				7	8	0

Table 14.6: Multiple-precision division after normalization (see Example 14.24).

14.25 Note (computational efficiency of Algorithm 14.20 with normalization)

- (i) (*multiplication count*) Assuming that normalization extends the number of digits in x by 1, each iteration of step 3 requires $1 + (t + 2) = t + 3$ single-precision multiplications. Hence, Algorithm 14.20 with normalization requires about $(n - t)(t + 3)$ single-precision multiplications.
- (ii) (*division count*) Since step 3.1 of Algorithm 14.20 is executed $n - t$ times, at most $n - t$ single-precision divisions are required when normalization is used.

14.3 Multiple-precision modular arithmetic

§14.2 provided methods for carrying out the basic operations (addition, subtraction, multiplication, squaring, and division) with multiple-precision integers. This section deals with these operations in \mathbb{Z}_m , the integers modulo m , where m is a multiple-precision positive integer. (See §2.4.3 for definitions of \mathbb{Z}_m and related operations.)

Let $m = (m_n m_{n-1} \cdots m_1 m_0)_b$ be a positive integer in radix b representation. Let $x = (x_n x_{n-1} \cdots x_1 x_0)_b$ and $y = (y_n y_{n-1} \cdots y_1 y_0)_b$ be non-negative integers in base b representation such that $x < m$ and $y < m$. Methods described in this section are for computing $x + y \bmod m$ (*modular addition*), $x - y \bmod m$ (*modular subtraction*), and $x \cdot y \bmod m$ (*modular multiplication*). Computing $x^{-1} \bmod m$ (*modular inversion*) is addressed in §14.4.3.

14.26 Definition If z is any integer, then $z \bmod m$ (the integer remainder in the range $[0, m - 1]$ after z is divided by m) is called the *modular reduction* of z with respect to modulus m .

Modular addition and subtraction

As is the case for ordinary multiple-precision operations, addition and subtraction are the simplest to compute of the modular operations.

14.27 Fact Let x and y be non-negative integers with $x, y < m$. Then:

- (i) $x + y < 2m$;
- (ii) if $x \geq y$, then $0 \leq x - y < m$; and
- (iii) if $x < y$, then $0 \leq x + m - y < m$.

If $x, y \in \mathbb{Z}_m$, then modular addition can be performed by using Algorithm 14.7 to add x and y as multiple-precision integers, with the additional step of subtracting m if (and only if) $x + y \geq m$. Modular subtraction is precisely Algorithm 14.9, provided $x \geq y$.

14.3.1 Classical modular multiplication

Modular multiplication is more involved than multiple-precision multiplication (§14.2.3), requiring both multiple-precision multiplication and some method for performing modular reduction (Definition 14.26). The most straightforward method for performing modular reduction is to compute the remainder on division by m , using a multiple-precision division algorithm such as Algorithm 14.20; this is commonly referred to as the *classical algorithm* for performing modular multiplication.

14.28 Algorithm Classical modular multiplication

INPUT: two positive integers x, y and a modulus m , all in radix b representation.

OUTPUT: $x \cdot y \bmod m$.

1. Compute $x \cdot y$ (using Algorithm 14.12).
2. Compute the remainder r when $x \cdot y$ is divided by m (using Algorithm 14.20).
3. Return(r).

14.3.2 Montgomery reduction

Montgomery reduction is a technique which allows efficient implementation of modular multiplication without explicitly carrying out the classical modular reduction step.

Let m be a positive integer, and let R and T be integers such that $R > m$, $\gcd(m, R) = 1$, and $0 \leq T < mR$. A method is described for computing $TR^{-1} \bmod m$ without using the classical method of Algorithm 14.28. $TR^{-1} \bmod m$ is called a *Montgomery reduction* of T modulo m with respect to R . With a suitable choice of R , a Montgomery reduction can be efficiently computed.

Suppose x and y are integers such that $0 \leq x, y < m$. Let $\tilde{x} = xR \bmod m$ and $\tilde{y} = yR \bmod m$. The Montgomery reduction of $\tilde{x}\tilde{y}$ is $\tilde{x}\tilde{y}R^{-1} \bmod m = xyR \bmod m$. This observation is used in Algorithm 14.94 to provide an efficient method for modular exponentiation.

To briefly illustrate, consider computing $x^5 \bmod m$ for some integer x , $1 \leq x < m$. First compute $\tilde{x} = xR \bmod m$. Then compute the Montgomery reduction of $\tilde{x}\tilde{x}$, which is $A = \tilde{x}^2 R^{-1} \bmod m$. The Montgomery reduction of A^2 is $A^2 R^{-1} \bmod m = \tilde{x}^4 R^{-3} \bmod m$. Finally, the Montgomery reduction of $(A^2 R^{-1} \bmod m)\tilde{x}$ is $(A^2 R^{-1})\tilde{x}R^{-1} \bmod m = \tilde{x}^5 R^{-4} \bmod m = x^5 R \bmod m$. Multiplying this value by $R^{-1} \bmod m$ and reducing

modulo m gives $x^5 \bmod m$. Provided that Montgomery reductions are more efficient to compute than classical modular reductions, this method may be more efficient than computing $x^5 \bmod m$ by repeated application of Algorithm 14.28.

If m is represented as a base b integer of length n , then a typical choice for R is b^n . The condition $R > m$ is clearly satisfied, but $\gcd(R, m) = 1$ will hold only if $\gcd(b, m) = 1$. Thus, this choice of R is not possible for all moduli. For those moduli of practical interest (such as RSA moduli), m will be odd; then b can be a power of 2 and $R = b^n$ will suffice.

Fact 14.29 is basic to the Montgomery reduction method. Note 14.30 then implies that $R = b^n$ is sufficient (but not necessary) for efficient implementation.

14.29 Fact (Montgomery reduction) Given integers m and R where $\gcd(m, R) = 1$, let $m' = -m^{-1} \bmod R$, and let T be any integer such that $0 \leq T < mR$. If $U = Tm' \bmod R$, then $(T + Um)/R$ is an integer and $(T + Um)/R \equiv TR^{-1} \pmod{m}$.

Justification. $T + Um \equiv T \pmod{m}$ and, hence, $(T + Um)R^{-1} \equiv TR^{-1} \pmod{m}$. To see that $(T + Um)R^{-1}$ is an integer, observe that $U = Tm' + kR$ and $m'm = -1 + lR$ for some integers k and l . It follows that $(T + Um)/R = (T + (Tm' + kR)m)/R = (T + T(-1 + lR) + kRm)/R = lT + km$.

14.30 Note (implications of Fact 14.29)

(i) $(T + Um)/R$ is an estimate for $TR^{-1} \bmod m$. Since $T < mR$ and $U < R$, then $(T + Um)/R < (mR + mR)/R = 2m$. Thus either $(T + Um)/R = TR^{-1} \bmod m$ or $(T + Um)/R = (TR^{-1} \bmod m) + m$ (i.e., the estimate is within m of the residue). Example 14.31 illustrates that both possibilities can occur.

(ii) If all integers are represented in radix b and $R = b^n$, then $TR^{-1} \bmod m$ can be computed with two multiple-precision multiplications (i.e., $U = T \cdot m'$ and $U \cdot m$) and simple right-shifts of $T + Um$ in order to divide by R .

14.31 Example (Montgomery reduction) Let $m = 187$, $R = 190$. Then $R^{-1} \bmod m = 125$, $m^{-1} \bmod R = 63$, and $m' = 127$. If $T = 563$, then $U = Tm' \bmod R = 61$ and $(T + Um)/R = 63 = TR^{-1} \bmod m$. If $T = 1125$ then $U = Tm' \bmod R = 185$ and $(T + Um)/R = 188 = (TR^{-1} \bmod m) + m$. \square

Algorithm 14.32 computes the Montgomery reduction of $T = (t_{2n-1} \cdots t_1 t_0)_b$ when $R = b^n$ and $m = (m_{n-1} \cdots m_1 m_0)_b$. The algorithm makes implicit use of Fact 14.29 by computing quantities which have similar properties to $U = Tm' \bmod R$ and $T + Um$, although the latter two expressions are not computed explicitly.

14.32 Algorithm Montgomery reduction

INPUT: integers $m = (m_{n-1} \cdots m_1 m_0)_b$ with $\gcd(m, b) = 1$, $R = b^n$, $m' = -m^{-1} \bmod b$, and $T = (t_{2n-1} \cdots t_1 t_0)_b < mR$.

OUTPUT: $TR^{-1} \bmod m$.

1. $A \leftarrow T$. (Notation: $A = (a_{2n-1} \cdots a_1 a_0)_b$.)
2. For i from 0 to $(n - 1)$ do the following:
 - 2.1 $u_i \leftarrow a_i m' \bmod b$.
 - 2.2 $A \leftarrow A + u_i m b^i$.
3. $A \leftarrow A / b^n$.
4. If $A \geq m$ then $A \leftarrow A - m$.
5. Return(A).

14.33 Note (comments on Montgomery reduction)

- (i) Algorithm 14.32 does not require $m' = -m^{-1} \bmod R$, as Fact 14.29 does, but rather $m' = -m^{-1} \bmod b$. This is due to the choice of $R = b^n$.
- (ii) At step 2.1 of the algorithm with $i = l$, A has the property that $a_j = 0, 0 \leq j \leq l-1$. Step 2.2 does not modify these values, but does replace a_l by 0. It follows that in step 3, A is divisible by b^n .
- (iii) Going into step 3, the value of A equals T plus some multiple of m (see step 2.2); here $A = (T + km)/b^n$ is an integer (see (ii) above) and $A \equiv TR^{-1} \pmod{m}$. It remains to show that A is less than $2m$, so that at step 4, a subtraction (rather than a division) will suffice. Going into step 3, $A = T + \sum_{i=0}^{n-1} u_i b^i m$. But $\sum_{i=0}^{n-1} u_i b^i m < b^n m = Rm$ and $T < Rm$; hence, $A < 2Rm$. Going into step 4 (after division of A by R), $A < 2m$ as required.

14.34 Note (computational efficiency of Montgomery reduction) Step 2.1 and step 2.2 of Algorithm 14.32 require a total of $n + 1$ single-precision multiplications. Since these steps are executed n times, the total number of single-precision multiplications is $n(n + 1)$. Algorithm 14.32 does not require any single-precision divisions.

14.35 Example (Montgomery reduction) Let $m = 72639$, $b = 10$, $R = 10^5$, and $T = 7118368$. Here $n = 5$, $m' = -m^{-1} \bmod 10 = 1$, $T \bmod m = 72385$, and $TR^{-1} \bmod m = 39796$. Table 14.7 displays the iterations of step 2 in Algorithm 14.32. \square

i	$u_i = a_i m' \bmod 10$	$u_i m b^i$	A
—	—	—	7118368
0	8	581112	7699480
1	8	5811120	13510600
2	6	43583400	57094000
3	4	290556000	347650000
4	5	3631950000	3979600000

Table 14.7: Montgomery reduction algorithm (see Example 14.35).

Montgomery multiplication

Algorithm 14.36 combines Montgomery reduction (Algorithm 14.32) and multiple-precision multiplication (Algorithm 14.12) to compute the Montgomery reduction of the product of two integers.

14.36 Algorithm Montgomery multiplication

INPUT: integers $m = (m_{n-1} \cdots m_1 m_0)_b$, $x = (x_{n-1} \cdots x_1 x_0)_b$, $y = (y_{n-1} \cdots y_1 y_0)_b$ with $0 \leq x, y < m$, $R = b^n$ with $\gcd(m, b) = 1$, and $m' = -m^{-1} \bmod b$.

OUTPUT: $xyR^{-1} \bmod m$.

1. $A \leftarrow 0$. (Notation: $A = (a_n a_{n-1} \cdots a_1 a_0)_b$.)
2. For i from 0 to $(n - 1)$ do the following:
 - 2.1 $u_i \leftarrow (a_0 + x_i y_0) m' \bmod b$.
 - 2.2 $A \leftarrow (A + x_i y + u_i m) / b$.
3. If $A \geq m$ then $A \leftarrow A - m$.
4. Return(A).

14.37 Note (*partial justification of Algorithm 14.36*) Suppose at the i^{th} iteration of step 2 that $0 \leq A < 2m - 1$. Step 2.2 replaces A with $(A + x_i y + u_i m)/b$; but $(A + x_i y + u_i m)/b \leq (2m - 2 + (b - 1)(m - 1) + (b - 1)m)/b = 2m - 1 - (1/b)$. Hence, $A < 2m - 1$, justifying step 3.

14.38 Note (*computational efficiency of Algorithm 14.36*) Since $A + x_i y + u_i m$ is a multiple of b , only a right-shift is required to perform a division by b in step 2.2. Step 2.1 requires two single-precision multiplications and step 2.2 requires $2n$. Since step 2 is executed n times, the total number of single-precision multiplications is $n(2 + 2n) = 2n(n + 1)$.

14.39 Note (*computing $xy \bmod m$ with Montgomery multiplication*) Suppose x , y , and m are n -digit base b integers with $0 \leq x, y < m$. Neglecting the cost of the precomputation in the input, Algorithm 14.36 computes $xyR^{-1} \bmod m$ with $2n(n + 1)$ single-precision multiplications. Neglecting the cost to compute $R^2 \bmod m$ and applying Algorithm 14.36 to $xyR^{-1} \bmod m$ and $R^2 \bmod m$, $xy \bmod m$ is computed in $4n(n + 1)$ single-precision operations. Using classical modular multiplication (Algorithm 14.28) would require $2n(n + 1)$ single-precision operations and no precomputation. Hence, the classical algorithm is superior for doing a single modular multiplication; however, Montgomery multiplication is very effective for performing modular exponentiation (Algorithm 14.94).

14.40 Remark (*Montgomery reduction vs. Montgomery multiplication*) Algorithm 14.36 (Montgomery multiplication) takes as input two n -digit numbers and then proceeds to interleave the multiplication and reduction steps. Because of this, Algorithm 14.36 is not able to take advantage of the special case where the input integers are equal (i.e., squaring). On the other hand, Algorithm 14.32 (Montgomery reduction) assumes as input the product of two integers, each of which has at most n digits. Since Algorithm 14.32 is independent of multiple-precision multiplication, a faster squaring algorithm such as Algorithm 14.16 may be used prior to the reduction step.

14.41 Example (*Montgomery multiplication*) In Algorithm 14.36, let $m = 72639$, $R = 10^5$, $x = 5792$, $y = 1229$. Here $n = 5$, $m' = -m^{-1} \bmod 10 = 1$, and $xyR^{-1} \bmod m = 39796$. Notice that m and R are the same values as in Example 14.35, as is $xy = 7118368$. Table 14.8 displays the steps in Algorithm 14.36. \square

i	x_i	$x_i y_0$	u_i	$x_i y$	$u_i m$	A
0	2	18	8	2458	581112	58357
1	9	81	8	11061	581112	65053
2	7	63	6	8603	435834	50949
3	5	45	4	6145	290556	34765
4	0	0	5	0	363195	39796

Table 14.8: Montgomery multiplication (see Example 14.41).

14.3.3 Barrett reduction

Barrett reduction (Algorithm 14.42) computes $r = x \bmod m$ given x and m . The algorithm requires the precomputation of the quantity $\mu = \lfloor b^{2k}/m \rfloor$; it is advantageous if many reductions are performed with a single modulus. For example, each RSA encryption for one entity requires reduction modulo that entity's public key modulus. The precomputation takes

a fixed amount of work, which is negligible in comparison to modular exponentiation cost. Typically, the radix b is chosen to be close to the word-size of the processor. Hence, assume $b > 3$ in Algorithm 14.42 (see Note 14.44 (ii)).

14.42 Algorithm Barrett modular reduction

INPUT: positive integers $x = (x_{2k-1} \cdots x_1 x_0)_b$, $m = (m_{k-1} \cdots m_1 m_0)_b$ (with $m_{k-1} \neq 0$), and $\mu = \lfloor b^{2k}/m \rfloor$.

OUTPUT: $r = x \bmod m$.

1. $q_1 \leftarrow \lfloor x/b^{k-1} \rfloor$, $q_2 \leftarrow q_1 \cdot \mu$, $q_3 \leftarrow \lfloor q_2/b^{k+1} \rfloor$.
2. $r_1 \leftarrow x \bmod b^{k+1}$, $r_2 \leftarrow q_3 \cdot m \bmod b^{k+1}$, $r \leftarrow r_1 - r_2$.
3. If $r < 0$ then $r \leftarrow r + b^{k+1}$.
4. While $r \geq m$ do: $r \leftarrow r - m$.
5. Return(r).

14.43 Fact By the division algorithm (Definition 2.82), there exist integers Q and R such that $x = Qm + R$ and $0 \leq R < m$. In step 1 of Algorithm 14.42, the following inequality is satisfied: $Q - 2 \leq q_3 \leq Q$.

14.44 Note (*partial justification of correctness of Barrett reduction*)

- (i) Algorithm 14.42 is based on the observation that $\lfloor x/m \rfloor$ can be written as $Q = \lfloor (x/b^{k-1})(b^{2k}/m)(1/b^{k+1}) \rfloor$. Moreover, Q can be approximated by the quantity $q_3 = \lfloor \lfloor x/b^{k-1} \rfloor \mu / b^{k+1} \rfloor$. Fact 14.43 guarantees that q_3 is never larger than the true quotient Q , and is at most 2 smaller.
- (ii) In step 2, observe that $-b^{k+1} < r_1 - r_2 < b^{k+1}$, $r_1 - r_2 \equiv (Q - q_3)m + R \pmod{b^{k+1}}$, and $0 \leq (Q - q_3)m + R < 3m < b^{k+1}$ since $m < b^k$ and $3 < b$. If $r_1 - r_2 \geq 0$, then $r_1 - r_2 = (Q - q_3)m + R$. If $r_1 - r_2 < 0$, then $r_1 - r_2 + b^{k+1} = (Q - q_3)m + R$. In either case, step 4 is repeated at most twice since $0 \leq r < 3m$.

14.45 Note (*computational efficiency of Barrett reduction*)

- (i) All divisions performed in Algorithm 14.42 are simple right-shifts of the base b representation.
- (ii) q_2 is only used to compute q_3 . Since the $k + 1$ least significant digits of q_2 are not needed to determine q_3 , only a partial multiple-precision multiplication (i.e., $q_1 \cdot \mu$) is necessary. The only influence of the $k + 1$ least significant digits on the higher order digits is the carry from position $k + 1$ to position $k + 2$. Provided the base b is sufficiently large with respect to k , this carry can be accurately computed by only calculating the digits at positions k and $k + 1$.¹ Hence, the $k - 1$ least significant digits of q_2 need not be computed. Since μ and q_1 have at most $k + 1$ digits, determining q_3 requires at most $(k + 1)^2 - \binom{k}{2} = (k^2 + 5k + 2)/2$ single-precision multiplications.
- (iii) In step 2 of Algorithm 14.42, r_2 can also be computed by a partial multiple-precision multiplication which evaluates only the least significant $k + 1$ digits of $q_3 \cdot m$. This can be done in at most $\binom{k+1}{2} + k$ single-precision multiplications.

14.46 Example (*Barrett reduction*) Let $b = 4$, $k = 3$, $x = (313221)_b$, and $m = (233)_b$ (i.e., $x = 3561$ and $m = 47$). Then $\mu = \lfloor 4^6/m \rfloor = 87 = (1113)_b$, $q_1 = \lfloor (313221)_b/4^2 \rfloor = (3132)_b$, $q_2 = (3132)_b \cdot (1113)_b = (10231302)_b$, $q_3 = (1023)_b$, $r_1 = (3221)_b$, $r_2 = (1023)_b \cdot (233)_b \bmod b^4 = (3011)_b$, and $r = r_1 - r_2 = (210)_b$. Thus $x \bmod m = 36$. \square

¹If $b > k$, then the carry computed by simply considering the digits at position $k - 1$ (and ignoring the carry from position $k - 2$) will be in error by at most 1.

14.3.4 Reduction methods for moduli of special form

When the modulus has a special (customized) form, reduction techniques can be employed to allow more efficient computation. Suppose that the modulus m is a t -digit base b positive integer of the form $m = b^t - c$, where c is an l -digit base b positive integer (for some $l < t$). Algorithm 14.47 computes $x \bmod m$ for any positive integer x by using only shifts, additions, and single-precision multiplications of base b numbers.

14.47 Algorithm Reduction modulo $m = b^t - c$

INPUT: a base b positive integer x and a modulus $m = b^t - c$, where c is an l -digit base b integer for some $l < t$.

OUTPUT: $r = x \bmod m$.

1. $q_0 \leftarrow \lfloor x/b^t \rfloor$, $r_0 \leftarrow x - q_0 b^t$, $r \leftarrow r_0$, $i \leftarrow 0$.
2. While $q_i > 0$ do the following:
 - 2.1 $q_{i+1} \leftarrow \lfloor q_i c / b^t \rfloor$, $r_{i+1} \leftarrow q_i c - q_{i+1} b^t$.
 - 2.2 $i \leftarrow i + 1$, $r \leftarrow r + r_i$.
3. While $r \geq m$ do: $r \leftarrow r - m$.
4. Return(r).

14.48 Example (reduction modulo $b^t - c$) Let $b = 4$, $m = 935 = (32213)_4$, and $x = 31085 = (13211231)_4$. Since $m = 4^5 - (1121)_4$, take $c = (1121)_4$. Here $t = 5$ and $l = 4$. Table 14.9 displays the quotients and remainders produced by Algorithm 14.47. At the beginning of step 3, $r = (102031)_4$. Since $r > m$, step 3 computes $r - m = (3212)_4$. \square

i	$q_{i-1}c$	q_i	r_i	r
0	—	$(132)_4$	$(11231)_4$	$(11231)_4$
1	$(221232)_4$	$(2)_4$	$(21232)_4$	$(33123)_4$
2	$(2302)_4$	$(0)_4$	$(2302)_4$	$(102031)_4$

Table 14.9: Reduction modulo $m = b^t - c$ (see Example 14.48).

14.49 Fact (termination) For some integer $s \geq 0$, $q_s = 0$; hence, Algorithm 14.47 terminates.

Justification. $q_i c = q_{i+1} b^t + r_{i+1}$, $i \geq 0$. Since $c < b^t$, $q_i = (q_{i+1} b^t / c) + (r_{i+1} / c) > q_{i+1}$. Since the q_i 's are non-negative integers which strictly decrease as i increases, there is some integer $s \geq 0$ such that $q_s = 0$.

14.50 Fact (correctness) Algorithm 14.47 terminates with the correct residue modulo m .

Justification. Suppose that s is the smallest index i for which $q_i = 0$ (i.e., $q_s = 0$). Now, $x = q_0 b^t + r_0$ and $q_i c = q_{i+1} b^t + r_{i+1}$, $0 \leq i \leq s-1$. Adding these equations gives $x + \left(\sum_{i=0}^{s-1} q_i\right) c = \left(\sum_{i=0}^{s-1} q_i\right) b^t + \sum_{i=0}^{s-1} r_i$. Since $b^t \equiv c \pmod{m}$, it follows that $x \equiv \sum_{i=0}^s r_i \pmod{m}$. Hence, repeated subtraction of m from $r = \sum_{i=0}^s r_i$ gives the correct residue.

14.51 Note (*computational efficiency of reduction modulo $b^t - c$*)

- (i) Suppose that x has $2t$ base b digits. If $l \leq t/2$, then Algorithm 14.47 executes step 2 at most $s = 3$ times, requiring 2 multiplications by c . In general, if l is approximately $(s - 2)t/(s - 1)$, then Algorithm 14.47 executes step 2 about s times. Thus, Algorithm 14.47 requires about sl single-precision multiplications.
- (ii) If c has few non-zero digits, then multiplication by c will be relatively inexpensive. If c is large but has few non-zero digits, the number of iterations of Algorithm 14.47 will be greater, but each iteration requires a very simple multiplication.

14.52 Note (*modifications*) Algorithm 14.47 can be modified if $m = b^t + c$ for some positive integer $c < b^t$: in step 2.2, replace $r \leftarrow r + r_i$ with $r \leftarrow r + (-1)^i r_i$.

14.53 Remark (*using moduli of a special form*) Selecting RSA moduli of the form $b^t \pm c$ for small values of c limits the choices of primes p and q . Care must also be exercised when selecting moduli of a special form, so that factoring is not made substantially easier; this is because numbers of this form are more susceptible to factoring by the special number field sieve (see §3.2.7). A similar statement can be made regarding the selection of primes of a special form for cryptographic schemes based on the discrete logarithm problem.

14.4 Greatest common divisor algorithms

Many situations in cryptography require the computation of the greatest common divisor (gcd) of two positive integers (see Definition 2.86). Algorithm 2.104 describes the classical Euclidean algorithm for this computation. For multiple-precision integers, Algorithm 2.104 requires a multiple-precision division at step 1.1 which is a relatively expensive operation. This section describes three methods for computing the gcd which are more efficient than the classical approach using multiple-precision numbers. The first is non-Euclidean and is referred to as the *binary gcd algorithm* (§14.4.1). Although it requires more steps than the classical algorithm, the binary gcd algorithm eliminates the computationally expensive division and replaces it with elementary shifts and additions. Lehmer's gcd algorithm (§14.4.2) is a variant of the classical algorithm more suited to multiple-precision computations. A binary version of the extended Euclidean algorithm is given in §14.4.3.

14.4.1 Binary gcd algorithm

14.54 Algorithm Binary gcd algorithm

INPUT: two positive integers x and y with $x \geq y$.

OUTPUT: $\gcd(x, y)$.

1. $g \leftarrow 1$.
 2. While both x and y are even do the following: $x \leftarrow x/2$, $y \leftarrow y/2$, $g \leftarrow 2g$.
 3. While $x \neq 0$ do the following:
 - 3.1 While x is even do: $x \leftarrow x/2$.
 - 3.2 While y is even do: $y \leftarrow y/2$.
 - 3.3 $t \leftarrow |x - y|/2$.
 - 3.4 If $x \geq y$ then $x \leftarrow t$; otherwise, $y \leftarrow t$.
 4. Return($g \cdot y$).
-

14.55 Example (*binary gcd algorithm*) The following table displays the steps performed by Algorithm 14.54 for computing $\gcd(1764, 868) = 28$. \square

x	1764	441	112	7	7	7	7	7	0
y	868	217	217	217	105	49	21	7	7
g	1	4	4	4	4	4	4	4	4

14.56 Note (*computational efficiency of Algorithm 14.54*)

- (i) If x and y are in radix 2 representation, then the divisions by 2 are simply right-shifts.
- (ii) Step 3.3 for multiple-precision integers can be computed using Algorithm 14.9.

14.4.2 Lehmer's gcd algorithm

Algorithm 14.57 is a variant of the classical Euclidean algorithm (Algorithm 2.104) and is suited to computations involving multiple-precision integers. It replaces many of the multiple-precision divisions by simpler single-precision operations.

Let x and y be positive integers in radix b representation, with $x \geq y$. Without loss of generality, assume that x and y have the same number of base b digits throughout Algorithm 14.57; this may necessitate padding the high-order digits of y with 0's.

14.57 Algorithm Lehmer's gcd algorithm

INPUT: two positive integers x and y in radix b representation, with $x \geq y$.

OUTPUT: $\gcd(x, y)$.

1. While $y \geq b$ do the following:
 - 1.1 Set \tilde{x}, \tilde{y} to be the high-order digit of x, y , respectively (\tilde{y} could be 0).
 - 1.2 $A \leftarrow 1, B \leftarrow 0, C \leftarrow 0, D \leftarrow 1$.
 - 1.3 While $(\tilde{y} + C) \neq 0$ and $(\tilde{y} + D) \neq 0$ do the following:
 - $q \leftarrow \lfloor (\tilde{x} + A) / (\tilde{y} + C) \rfloor, q' \leftarrow \lfloor (\tilde{x} + B) / (\tilde{y} + D) \rfloor$.
 - If $q \neq q'$ then go to step 1.4.
 - $t \leftarrow A - qC, A \leftarrow C, C \leftarrow t, t \leftarrow B - qD, B \leftarrow D, D \leftarrow t$.
 - $t \leftarrow \tilde{x} - q\tilde{y}, \tilde{x} \leftarrow \tilde{y}, \tilde{y} \leftarrow t$.
 - 1.4 If $B = 0$, then $T \leftarrow x \bmod y, x \leftarrow y, y \leftarrow T$;
otherwise, $T \leftarrow Ax + By, u \leftarrow Cx + Dy, x \leftarrow T, y \leftarrow u$.
2. Compute $v = \gcd(x, y)$ using Algorithm 2.104.
3. Return(v).

14.58 Note (*implementation notes for Algorithm 14.57*)

- (i) T is a multiple-precision variable. A, B, C, D , and t are signed single-precision variables; hence, one bit of each of these variables must be reserved for the sign.
- (ii) The first operation of step 1.3 may result in overflow since $0 \leq \tilde{x} + A, \tilde{y} + D \leq b$. This possibility needs to be accommodated. One solution is to reserve two bits more than the number of bits in a digit for each of \tilde{x} and \tilde{y} to accommodate both the sign and the possible overflow.
- (iii) The multiple-precision additions of step 1.4 are actually subtractions, since $AB \leq 0$ and $CD \leq 0$.

14.59 Note (*computational efficiency of Algorithm 14.57*)

- (i) Step 1.3 attempts to simulate multiple-precision divisions by much simpler single-precision operations. In each iteration of step 1.3, all computations are single precision. The number of iterations of step 1.3 depends on b .
- (ii) The modular reduction in step 1.4 is a multiple-precision operation. The other operations are multiple-precision, but require only linear time since the multipliers are single precision.

14.60 Example (*Lehmer's gcd algorithm*) Let $b = 10^3$, $x = 768\,454\,923$, and $y = 542\,167\,814$. Since $b = 10^3$, the high-order digits of x and y are $\tilde{x} = 768$ and $\tilde{y} = 542$, respectively. Table 14.10 displays the values of the variables at various stages of Algorithm 14.57. The single-precision computations (Step 1.3) when $q = q'$ are shown in Table 14.11. Hence $\gcd(x, y) = 1$. \square

14.4.3 Binary extended gcd algorithm

Given integers x and y , Algorithm 2.107 computes integers a and b such that $ax + by = v$, where $v = \gcd(x, y)$. It has the drawback of requiring relatively costly multiple-precision divisions when x and y are multiple-precision integers. Algorithm 14.61 eliminates this requirement at the expense of more iterations.

14.61 Algorithm Binary extended gcd algorithm

INPUT: two positive integers x and y .

OUTPUT: integers a , b , and v such that $ax + by = v$, where $v = \gcd(x, y)$.

1. $g \leftarrow 1$.
2. While x and y are both even, do the following: $x \leftarrow x/2$, $y \leftarrow y/2$, $g \leftarrow 2g$.
3. $u \leftarrow x$, $v \leftarrow y$, $A \leftarrow 1$, $B \leftarrow 0$, $C \leftarrow 0$, $D \leftarrow 1$.
4. While u is even do the following:
 - 4.1. $u \leftarrow u/2$.
 - 4.2. If $A \equiv B \equiv 0 \pmod{2}$ then $A \leftarrow A/2$, $B \leftarrow B/2$; otherwise, $A \leftarrow (A + y)/2$, $B \leftarrow (B - x)/2$.
5. While v is even do the following:
 - 5.1. $v \leftarrow v/2$.
 - 5.2. If $C \equiv D \equiv 0 \pmod{2}$ then $C \leftarrow C/2$, $D \leftarrow D/2$; otherwise, $C \leftarrow (C + y)/2$, $D \leftarrow (D - x)/2$.
6. If $u \geq v$ then $u \leftarrow u - v$, $A \leftarrow A - C$, $B \leftarrow B - D$; otherwise, $v \leftarrow v - u$, $C \leftarrow C - A$, $D \leftarrow D - B$.
7. If $u = 0$, then $a \leftarrow C$, $b \leftarrow D$, and return $(a, b, g \cdot v)$; otherwise, go to step 4.

14.62 Example (*binary extended gcd algorithm*) Let $x = 693$ and $y = 609$. Table 14.12 displays the steps in Algorithm 14.61 for computing integers a , b , v such that $693a + 609b = v$, where $v = \gcd(693, 609)$. The algorithm returns $v = 21$, $a = -181$, and $b = 206$. \square

x	y	q	q'	precision	reference
768 454 923	542 167 814	1	1	single	Table 14.11(i)
89 593 596	47 099 917	1	1	single	Table 14.11(ii)
42 493 679	4 606 238	10	8	multiple	
4 606 238	1 037 537	5	2	multiple	
1 037 537	456 090	—	—	multiple	
456 090	125 357	3	3	single	Table 14.11(iii)
34 681	10 657	3	3	single	Table 14.11(iv)
10 657	2 710	5	3	multiple	
2 710	2 527	1	0	multiple	
2 527	183				Algorithm 2.104
183	148				Algorithm 2.104
148	35				Algorithm 2.104
35	8				Algorithm 2.104
8	3				Algorithm 2.104
3	2				Algorithm 2.104
2	1				Algorithm 2.104
1	0				Algorithm 2.104

Table 14.10: Lehmer's gcd algorithm (see Example 14.60).

	\tilde{x}	\tilde{y}	A	B	C	D	q	q'
(i)	768	542	1	0	0	1	1	1
	542	226	0	1	1	-1	2	2
	226	90	1	-1	-2	3	2	2
	90	46	-2	3	5	-7	1	2
(ii)	89	47	1	0	0	1	1	1
	47	42	0	1	1	-1	1	1
	42	5	1	-1	-1	2	10	5
(iii)	456	125	1	0	0	1	3	3
	125	81	0	1	1	-3	1	1
	81	44	1	-3	-1	4	1	1
	44	37	-1	4	2	-7	1	1
	37	7	2	-7	-3	11	9	1
(iv)	34	10	1	0	0	1	3	3
	10	4	0	1	1	-3	2	11

Table 14.11: Single-precision computations (see Example 14.60 and Table 14.10).

u	v	A	B	C	D
693	609	1	0	0	1
84	609	1	-1	0	1
42	609	305	-347	0	1
21	609	457	-520	0	1
21	588	457	-520	-457	521
21	294	457	-520	76	-86
21	147	457	-520	38	-43
21	126	457	-520	-419	477
21	63	457	-520	95	-108
21	42	457	-520	-362	412
21	21	457	-520	-181	206
0	21	638	-726	-181	206

Table 14.12: The binary extended gcd algorithm with $x = 693$, $y = 609$ (see Example 14.62).

14.63 Note (computational efficiency of Algorithm 14.61)

- (i) The only multiple-precision operations needed for Algorithm 14.61 are addition and subtraction. Division by 2 is simply a right-shift of the binary representation.
- (ii) The number of bits needed to represent either u or v decreases by (at least) 1, after at most two iterations of steps 4 – 7; thus, the algorithm takes at most $2(\lfloor \lg x \rfloor + \lfloor \lg y \rfloor + 2)$ such iterations.

14.64 Note (multiplicative inverses) Given positive integers m and a , it is often necessary to find an integer $z \in \mathbb{Z}_m$ such that $az \equiv 1 \pmod{m}$, if such an integer exists. z is called the multiplicative inverse of a modulo m (see Definition 2.115). For example, constructing the private key for RSA requires the computation of an integer d such that $ed \equiv 1 \pmod{(p-1)(q-1)}$ (see Algorithm 8.1). Algorithm 14.61 provides a computationally efficient method for determining z given a and m , by setting $x = m$ and $y = a$. If $\gcd(x, y) = 1$, then, at termination, $z = D$ if $D > 0$, or $z = m - D$ if $D < 0$; if $\gcd(x, y) \neq 1$, then a is not invertible modulo m . Notice that if m is odd, it is not necessary to compute the values of A and C . It would appear that step 4 of Algorithm 14.61 requires both A and B in order to decide which case in step 4.2 is executed. But if m is odd and B is even, then A must be even; hence, the decision can be made using the parities of B and m .

Example 14.65 illustrates Algorithm 14.61 for computing a multiplicative inverse.

14.65 Example (multiplicative inverse) Let $m = 383$ and $a = 271$. Table 14.13 illustrates the steps of Algorithm 14.61 for computing $271^{-1} \bmod 383 = 106$. Notice that values for the variables A and C need not be computed. \square

14.5 Chinese remainder theorem for integers

Fact 2.120 introduced the Chinese remainder theorem (CRT) and Fact 2.121 outlined an algorithm for solving the associated system of linear congruences. Although the method described there is the one found in most textbooks on elementary number theory, it is not the

iteration:	1	2	3	4	5	6	7	8	9	10
u	383	112	56	28	14	7	7	7	7	7
v	271	271	271	271	271	271	264	132	66	33
B	0	-1	-192	-96	-48	-24	-24	-24	-24	-24
D	1	1	1	1	1	1	25	-179	-281	-332
iteration:	11	12	13	14	15	16	17	18	19	
u	7	7	7	7	4	2	1	1	1	
v	26	13	6	3	3	3	3	2	1	
B	-24	-24	-24	-24	41	-171	-277	-277	-277	
D	-308	-154	-130	-65	-65	-65	-65	212	106	

Table 14.13: Inverse computation using the binary extended gcd algorithm (see Example 14.65).

method of choice for large integers. Garner's algorithm (Algorithm 14.71) has some computational advantages. §14.5.1 describes an alternate (non-radix) representation for non-negative integers, called a *modular representation*, that allows some computational advantages compared to standard radix representations. Algorithm 14.71 provides a technique for converting numbers from modular to base b representation.

14.5.1 Residue number systems

In previous sections, non-negative integers have been represented in radix b notation. An alternate means is to use a mixed-radix representation.

14.66 Fact Let B be a fixed positive integer. Let m_1, m_2, \dots, m_t be positive integers such that $\gcd(m_i, m_j) = 1$ for all $i \neq j$, and $M = \prod_{i=1}^t m_i \geq B$. Then each integer x , $0 \leq x < B$, can be uniquely represented by the sequence of integers $v(x) = (v_1, v_2, \dots, v_t)$, where $v_i = x \bmod m_i$, $1 \leq i \leq t$.

14.67 Definition Referring to Fact 14.66, $v(x)$ is called the *modular representation* or *mixed-radix representation* of x for the moduli m_1, m_2, \dots, m_t . The set of modular representations for all integers x in the range $0 \leq x < B$ is called a *residue number system*.

If $v(x) = (v_1, v_2, \dots, v_t)$ and $v(y) = (u_1, u_2, \dots, u_t)$, define $v(x) + v(y) = (w_1, w_2, \dots, w_t)$ where $w_i = v_i + u_i \bmod m_i$, and $v(x) \cdot v(y) = (z_1, z_2, \dots, z_t)$ where $z_i = v_i \cdot u_i \bmod m_i$.

14.68 Fact If $0 \leq x, y < M$, then $v((x + y) \bmod M) = v(x) + v(y)$ and $v((x \cdot y) \bmod M) = v(x) \cdot v(y)$.

14.69 Example (modular representation) Let $M = 30 = 2 \times 3 \times 5$; here, $t = 3$, $m_1 = 2$, $m_2 = 3$, and $m_3 = 5$. Table 14.14 displays each residue modulo 30 along with its associated modular representation. As an example of Fact 14.68, note that $21 + 27 \equiv 18 \pmod{30}$ and $(101) + (102) = (003)$. Also $22 \cdot 17 \equiv 14 \pmod{30}$ and $(012) \cdot (122) = (024)$. \square

14.70 Note (computational efficiency of modular representation for RSA decryption) Suppose that $n = pq$, where p and q are distinct primes. Fact 14.68 implies that $x^d \bmod n$ can be computed in a modular representation as $v^d(x)$; that is, if $v(x) = (v_1, v_2)$ with respect to moduli $m_1 = p$, $m_2 = q$, then $v^d(x) = (v_1^d \bmod p, v_2^d \bmod q)$. In general, computing

x	$v(x)$	x	$v(x)$	x	$v(x)$	x	$v(x)$	x	$v(x)$
0	(000)	6	(001)	12	(002)	18	(003)	24	(004)
1	(111)	7	(112)	13	(113)	19	(114)	25	(110)
2	(022)	8	(023)	14	(024)	20	(020)	26	(021)
3	(103)	9	(104)	15	(100)	21	(101)	27	(102)
4	(014)	10	(010)	16	(011)	22	(012)	28	(013)
5	(120)	11	(121)	17	(122)	23	(123)	29	(124)

Table 14.14: Modular representations (see Example 14.69).

$v_1^d \bmod p$ and $v_2^d \bmod q$ is faster than computing $x^d \bmod n$. For RSA, if p and q are part of the private key, modular representation can be used to improve the performance of both decryption and signature generation (see Note 14.75).

Converting an integer x from a base b representation to a modular representation is easily done by applying a modular reduction algorithm to compute $v_i = x \bmod m_i$, $1 \leq i \leq t$. Modular representations of integers in \mathbb{Z}_M may facilitate some computational efficiencies, provided conversion from a standard radix to modular representation and back are relatively efficient operations. Algorithm 14.71 describes one way of converting from modular representation back to a standard radix representation.

14.5.2 Garner's algorithm

Garner's algorithm is an efficient method for determining x , $0 \leq x < M$, given $v(x) = (v_1, v_2, \dots, v_t)$, the residues of x modulo the pairwise co-prime moduli m_1, m_2, \dots, m_t .

14.71 Algorithm Garner's algorithm for CRT

INPUT: a positive integer $M = \prod_{i=1}^t m_i > 1$, with $\gcd(m_i, m_j) = 1$ for all $i \neq j$, and a modular representation $v(x) = (v_1, v_2, \dots, v_t)$ of x for the m_i .

OUTPUT: the integer x in radix b representation.

1. For i from 2 to t do the following:
 - 1.1 $C_i \leftarrow 1$.
 - 1.2 For j from 1 to $(i-1)$ do the following:

$$u \leftarrow m_j^{-1} \bmod m_i \text{ (use Algorithm 14.61).}$$

$$C_i \leftarrow u \cdot C_i \bmod m_i.$$
2. $u \leftarrow v_1$, $x \leftarrow u$.
3. For i from 2 to t do the following: $u \leftarrow (v_i - x)C_i \bmod m_i$, $x \leftarrow x + u \cdot \prod_{j=1}^{i-1} m_j$.
4. Return(x).

14.72 Fact x returned by Algorithm 14.71 satisfies $0 \leq x < M$, $x \equiv v_i \pmod{m_i}$, $1 \leq i \leq t$.

14.73 Example (Garner's algorithm) Let $m_1 = 5$, $m_2 = 7$, $m_3 = 11$, $m_4 = 13$, $M = \prod_{i=1}^4 m_i = 5005$, and $v(x) = (2, 1, 3, 8)$. The constants C_i computed are $C_2 = 3$, $C_3 = 6$, and $C_4 = 5$. The values of (i, u, x) computed in step 3 of Algorithm 14.71 are $(1, 2, 2)$, $(2, 4, 22)$, $(3, 7, 267)$, and $(4, 5, 2192)$. Hence, the modular representation $v(x) = (2, 1, 3, 8)$ corresponds to the integer $x = 2192$. \square

14.74 Note (*computational efficiency of Algorithm 14.71*)

- (i) If Garner's algorithm is used repeatedly with the same modulus M and the same factors of M , then step 1 can be considered as a precomputation, requiring the storage of $t - 1$ numbers.
- (ii) The classical algorithm for the CRT (Algorithm 2.121) typically requires a modular reduction with modulus M , whereas Algorithm 14.71 does not. Suppose M is a kt -bit integer and each m_i is a k -bit integer. A modular reduction by M takes $O((kt)^2)$ bit operations, whereas a modular reduction by m_i takes $O(k^2)$ bit operations. Since Algorithm 14.71 only does modular reduction with m_i , $2 \leq i \leq t$, it takes $O(tk^2)$ bit operations in total for the reduction phase, and is thus more efficient.

14.75 Note (*RSA decryption and signature generation*)

- (i) (*special case of two moduli*) Algorithm 14.71 is particularly efficient for RSA moduli $n = pq$, where $m_1 = p$ and $m_2 = q$ are distinct primes. Step 1 computes a single value $C_2 = p^{-1} \bmod q$. Step 3 is executed once: $u = (v_2 - v_1)C_2 \bmod q$ and $x = v_1 + up$.
- (ii) (*RSA exponentiation*) Suppose p and q are t -bit primes, and let $n = pq$. Let d be a $2t$ -bit RSA private key. RSA decryption and signature generation compute $x^d \bmod n$ for some $x \in \mathbb{Z}_n$. Suppose that modular multiplication and squaring require k^2 bit operations for k -bit inputs, and that exponentiation with a k -bit exponent requires about $\frac{3}{2}k$ multiplications and squarings (see Note 14.78). Then computing $x^d \bmod n$ requires about $\frac{3}{2}(2t)^3 = 12t^3$ bit operations. A more efficient approach is to compute $x^{d_p} \bmod p$ and $x^{d_q} \bmod q$ (where $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$), and then use Garner's algorithm to construct $x^d \bmod pq$. Although this procedure takes two exponentiations, each is considerably more efficient because the moduli are smaller. Assuming that the cost of Algorithm 14.71 is negligible with respect to the exponentiations, computing $x^d \bmod n$ is about $\frac{3}{2}(2t)^3 / 2(\frac{3}{2}t^3) = 4$ times faster.

14.6 Exponentiation

One of the most important arithmetic operations for public-key cryptography is exponentiation. The RSA scheme (§8.2) requires exponentiation in \mathbb{Z}_m for some positive integer m , whereas Diffie-Hellman key agreement (§12.6.1) and the ElGamal encryption scheme (§8.4) use exponentiation in \mathbb{Z}_p for some large prime p . As pointed out in §8.4.2, ElGamal encryption can be generalized to any finite cyclic group. This section discusses methods for computing the *exponential* g^e , where the *base* g is an element of a finite group G (§2.5.1) and the *exponent* e is a non-negative integer. A reader uncomfortable with the setting of a general group may consider G to be \mathbb{Z}_m^* ; that is, read g^e as $g^e \bmod m$.

An efficient method for multiplying two elements in the group G is essential to performing efficient exponentiation. The most naive way to compute g^e is to do $e - 1$ multiplications in the group G . For cryptographic applications, the order of the group G typically exceeds 2^{160} elements, and may exceed 2^{1024} . Most choices of e are large enough that it would be infeasible to compute g^e using $e - 1$ successive multiplications by g .

There are two ways to reduce the time required to do exponentiation. One way is to decrease the time to multiply two elements in the group; the other is to reduce the number of multiplications used to compute g^e . Ideally, one would do both.

This section considers three types of exponentiation algorithms.

1. *basic techniques for exponentiation.* Arbitrary choices of the base g and exponent e are allowed.
2. *fixed-exponent exponentiation algorithms.* The exponent e is fixed and arbitrary choices of the base g are allowed. RSA encryption and decryption schemes benefit from such algorithms.
3. *fixed-base exponentiation algorithms.* The base g is fixed and arbitrary choices of the exponent e are allowed. ElGamal encryption and signatures schemes and Diffie-Hellman key agreement protocols benefit from such algorithms.

14.6.1 Techniques for general exponentiation

This section includes general-purpose exponentiation algorithms referred to as *repeated square-and-multiply* algorithms.

(i) Basic binary and k-ary exponentiation

Algorithm 14.76 is simply Algorithm 2.143 restated in terms of an arbitrary finite abelian group G with identity element 1.

14.76 Algorithm Right-to-left binary exponentiation

INPUT: an element $g \in G$ and integer $e \geq 1$.

OUTPUT: g^e .

1. $A \leftarrow 1, S \leftarrow g$.
2. While $e \neq 0$ do the following:
 - 2.1 If e is odd then $A \leftarrow A \cdot S$.
 - 2.2 $e \leftarrow \lfloor e/2 \rfloor$.
 - 2.3 If $e \neq 0$ then $S \leftarrow S \cdot S$.
3. Return(A).

14.77 Example (*right-to-left binary exponentiation*) The following table displays the values of A , e , and S during each iteration of Algorithm 14.76 for computing g^{283} . \square

A	1	g	g^3	g^3	g^{11}	g^{27}	g^{27}	g^{27}	g^{27}	g^{283}
e	283	141	70	35	17	8	4	2	1	0
S	g	g^2	g^4	g^8	g^{16}	g^{32}	g^{64}	g^{128}	g^{256}	—

14.78 Note (*computational efficiency of Algorithm 14.76*) Let $t + 1$ be the bitlength of the binary representation of e , and let $\text{wt}(e)$ be the number of 1's in this representation. Algorithm 14.76 performs t squarings and $\text{wt}(e) - 1$ multiplications. If e is randomly selected in the range $0 \leq e < |G| = n$, then about $\lfloor \lg n \rfloor$ squarings and $\frac{1}{2}(\lfloor \lg n \rfloor + 1)$ multiplications can be expected. (The assignment $1 \cdot x$ is not counted as a multiplication, nor is the operation $1 \cdot 1$ counted as a squaring.) If squaring is approximately as costly as an arbitrary multiplication (cf. Note 14.18), then the expected amount of work is roughly $\frac{3}{2} \lfloor \lg n \rfloor$ multiplications.

Algorithm 14.76 computes $A \cdot S$ whenever e is odd. For some choices of g , $A \cdot g$ can be computed more efficiently than $A \cdot S$ for arbitrary S . Algorithm 14.79 is a left-to-right binary exponentiation which replaces the operation $A \cdot S$ (for arbitrary S) by the operation $A \cdot g$ (for fixed g).

14.79 Algorithm Left-to-right binary exponentiationINPUT: $g \in G$ and a positive integer $e = (e_t e_{t-1} \cdots e_1 e_0)_2$.OUTPUT: g^e .

1. $A \leftarrow 1$.
2. For i from t down to 0 do the following:
 - 2.1 $A \leftarrow A \cdot A$.
 - 2.2 If $e_i = 1$, then $A \leftarrow A \cdot g$.
3. Return(A).

14.80 Example (*left-to-right binary exponentiation*) The following table displays the values of A during each iteration of Algorithm 14.79 for computing g^{283} . Note that $t = 8$ and $283 = (100011011)_2$. \square

i	8	7	6	5	4	3	2	1	0
e_i	1	0	0	0	1	1	0	1	1
A	g	g^2	g^4	g^8	g^{17}	g^{35}	g^{70}	g^{141}	g^{283}

14.81 Note (*computational efficiency of Algorithm 14.79*) Let $t + 1$ be the bitlength of the binary representation of e , and let $\text{wt}(e)$ be the number of 1's in this representation. Algorithm 14.79 performs $t + 1$ squarings and $\text{wt}(e) - 1$ multiplications by g . The number of squarings and multiplications is the same as in Algorithm 14.76 but, in this algorithm, multiplication is always with the fixed value g . If g has a special structure, this multiplication may be substantially easier than multiplying two arbitrary elements. For example, a frequent operation in ElGamal public-key schemes is the computation of $g^k \bmod p$, where g is a generator of \mathbb{Z}_p^* and p is a large prime number. The multiple-precision computation $A \cdot g$ can be done in linear time if g is chosen so that it can be represented by a single-precision integer (e.g., $g = 2$). If the radix b is sufficiently large, there is a high probability that such a generator exists.

Algorithm 14.82, sometimes referred to as the *window method* for exponentiation, is a generalization of Algorithm 14.79 which processes more than one bit of the exponent per iteration.

14.82 Algorithm Left-to-right k -ary exponentiationINPUT: g and $e = (e_t e_{t-1} \cdots e_1 e_0)_b$, where $b = 2^k$ for some $k \geq 1$.OUTPUT: g^e .

1. *Precomputation.*
 - 1.1 $g_0 \leftarrow 1$.
 - 1.2 For i from 1 to $(2^k - 1)$ do: $g_i \leftarrow g_{i-1} \cdot g$. (Thus, $g_i = g^i$.)
2. $A \leftarrow 1$.
3. For i from t down to 0 do the following:
 - 3.1 $A \leftarrow A^{2^k}$.
 - 3.2 $A \leftarrow A \cdot g_{e_i}$.
4. Return(A).

In Algorithm 14.83, Algorithm 14.82 is modified slightly to reduce the amount of precomputation. The following notation is used: for each i , $0 \leq i \leq t$, if $e_i \neq 0$, then write $e_i = 2^{h_i} u_i$ where u_i is odd; if $e_i = 0$, then let $h_i = 0$ and $u_i = 0$.

14.83 Algorithm Modified left-to-right k -ary exponentiation

INPUT: g and $e = (e_t e_{t-1} \cdots e_1 e_0)_b$, where $b = 2^k$ for some $k \geq 1$.

OUTPUT: g^e .

1. *Precomputation.*
 - 1.1 $g_0 \leftarrow 1, g_1 \leftarrow g, g_2 \leftarrow g^2$.
 - 1.2 For i from 1 to $(2^{k-1} - 1)$ do: $g_{2i+1} \leftarrow g_{2i-1} \cdot g_2$.
 2. $A \leftarrow 1$.
 3. For i from t down to 0 do: $A \leftarrow (A^{2^{k-h_i}} \cdot g_{u_i})^{2^{h_i}}$.
 4. Return(A).
-

14.84 Remark (*right-to-left k -ary exponentiation*) Algorithm 14.82 is a generalization of Algorithm 14.79. In a similar manner, Algorithm 14.76 can be generalized to the k -ary case. However, the optimization given in Algorithm 14.83 is not possible for the generalized right-to-left k -ary exponentiation method.

(ii) Sliding-window exponentiation

Algorithm 14.85 also reduces the amount of precomputation compared to Algorithm 14.82 and, moreover, reduces the average number of multiplications performed (excluding squarings). k is called the *window size*.

14.85 Algorithm Sliding-window exponentiation

INPUT: $g, e = (e_t e_{t-1} \cdots e_1 e_0)_2$ with $e_t = 1$, and an integer $k \geq 1$.

OUTPUT: g^e .

1. *Precomputation.*
 - 1.1 $g_1 \leftarrow g, g_2 \leftarrow g^2$.
 - 1.2 For i from 1 to $(2^{k-1} - 1)$ do: $g_{2i+1} \leftarrow g_{2i-1} \cdot g_2$.
 2. $A \leftarrow 1, i \leftarrow t$.
 3. While $i \geq 0$ do the following:
 - 3.1 If $e_i = 0$ then do: $A \leftarrow A^2, i \leftarrow i - 1$.
 - 3.2 Otherwise ($e_i \neq 0$), find the longest bitstring $e_i e_{i-1} \cdots e_l$ such that $i - l + 1 \leq k$ and $e_l = 1$, and do the following:

$$A \leftarrow A^{2^{i-l+1}} \cdot g_{(e_i e_{i-1} \cdots e_l)_2}, i \leftarrow l - 1.$$
 4. Return(A).
-

14.86 Example (*sliding-window exponentiation*) Take $e = 11749 = (10110111100101)_2$ and $k = 3$. Table 14.15 illustrates the steps of Algorithm 14.85. Notice that the sliding-window method for this exponent requires three multiplications, corresponding to $i = 7, 4$, and 0. Algorithm 14.79 would have required four multiplications for the same values of k and e . \square

i	A	Longest bitstring
13	1	101
10	g^5	101
7	$(g^5)^8 g^5 = g^{45}$	111
4	$(g^{45})^8 g^7 = g^{367}$	—
3	$(g^{367})^2 = g^{734}$	—
2	$(g^{734})^2 = g^{1468}$	101
0	$(g^{1468})^8 g^5 = g^{11749}$	—

Table 14.15: Sliding-window exponentiation with $k = 3$ and exponent $e = (10110111100101)_2$.

14.87 Note (comparison of exponentiation algorithms) Let $t + 1$ be the bitlength of e , and let $l + 1$ be the number of k -bit words formed from e ; that is, $l = \lceil (t + 1)/k \rceil - 1 = \lfloor t/k \rfloor$. Table 14.16 summarizes the number of squarings and multiplications required by Algorithms 14.76, 14.79, 14.82, and 14.83. Analysis of the number of squarings and multiplications for Algorithm 14.85 is more difficult, although it is the recommended method.

- (i) (*squarings for Algorithm 14.82*) The number of squarings for Algorithm 14.82 is lk . Observe that $lk = \lfloor t/k \rfloor k = t - (t \bmod k)$. It follows that $t - (k - 1) \leq lk \leq t$ and that Algorithm 14.82 can save up to $k - 1$ squarings over Algorithms 14.76 and 14.79. An optimal value for k in Algorithm 14.82 will depend on t .
- (ii) (*squarings for Algorithm 14.83*) The number of squarings for Algorithm 14.83 is $lk + h_l$ where $0 \leq h_l \leq t \bmod k$. Since $t - (k - 1) \leq lk \leq lk + h_l \leq lk + (t \bmod k) = t$ or $t - (k - 1) \leq lk + h_l \leq t$, the number of squarings for this algorithm has the same bounds as Algorithm 14.82.

Algorithm	Precomputation		squarings	Multiplications	
	sq	mult		worst case	average case
14.76	0	0	t	t	$t/2$
14.79	0	0	t	t	$t/2$
14.82	1	$2^k - 3$	$t - (k - 1) \leq lk \leq t$	$l - 1$	$l(2^k - 1)/2^k$
14.83	1	$2^{k-1} - 1$	$t - (k - 1) \leq lk + h_l \leq t$	$l - 1$	$l(2^k - 1)/2^k$

Table 14.16: Number of squarings (sq) and multiplications (mult) for exponentiation algorithms.

(iii) Simultaneous multiple exponentiation

There are a number of situations which require computation of the product of several exponentials with distinct bases and distinct exponents (for example, verification of ElGamal signatures; see Note 14.91). Rather than computing each exponential separately, Algorithm 14.88 presents a method to do them simultaneously.

Let e_0, e_1, \dots, e_{k-1} be positive integers each of bitlength t ; some of the high-order bits of some of the exponents might be 0, but there is at least one e_i whose high-order bit is 1. Form a $k \times t$ array EA (called the *exponent array*) whose rows are the binary representations of the exponents e_i , $0 \leq i \leq k - 1$. Let I_j be the non-negative integer whose binary representation is the j th column, $1 \leq j \leq t$, of EA , where low-order bits are at the top of the column.

14.88 Algorithm Simultaneous multiple exponentiation

INPUT: group elements g_0, g_1, \dots, g_{k-1} and non-negative t -bit integers e_0, e_1, \dots, e_{k-1} .
 OUTPUT: $g_0^{e_0} g_1^{e_1} \dots g_{k-1}^{e_{k-1}}$.

1. *Precomputation.* For i from 0 to $(2^k - 1)$: $G_i \leftarrow \prod_{j=0}^{k-1} g_j^{i_j}$ where $i = (i_{k-1} \dots i_0)_2$.
2. $A \leftarrow 1$.
3. For i from 1 to t do the following: $A \leftarrow A \cdot A$, $A \leftarrow A \cdot G_{I_i}$.
4. Return(A).

14.89 Example (*simultaneous multiple exponentiation*) In this example, $g_0^{30} g_1^{10} g_2^{24}$ is computed using Algorithm 14.88. Let $e_0 = 30 = (11110)_2$, $e_1 = 10 = (01010)_2$, and $e_2 = 24 = (11000)_2$. The 3×5 array EA is:

1	1	1	1	0
0	1	0	1	0
1	1	0	0	0

The next table displays precomputed values from step 1 of Algorithm 14.88.

i	0	1	2	3	4	5	6	7
G_i	1	g_0	g_1	$g_0 g_1$	g_2	$g_0 g_2$	$g_1 g_2$	$g_0 g_1 g_2$

Finally, the value of A at the end of each iteration of step 3 is shown in the following table. Here, $I_1 = 5$, $I_2 = 7$, $I_3 = 1$, $I_4 = 3$, and $I_5 = 0$.

i	1	2	3	4	5
A	$g_0 g_2$	$g_0^3 g_1 g_2^3$	$g_0^7 g_1^2 g_2^6$	$g_0^{15} g_1^5 g_2^{12}$	$g_0^{30} g_1^{10} g_2^{24}$

□

14.90 Note (*computational efficiency of Algorithm 14.88*)

- (i) Algorithm 14.88 computes $g_0^{e_0} g_1^{e_1} \dots g_{k-1}^{e_{k-1}}$ (where each e_i is represented by t bits) by performing $t - 1$ squarings and at most $(2^k - 2) + t - 1$ multiplications. The multiplication is trivial for any column consisting of all 0's.
- (ii) Not all of the G_i , $0 \leq i \leq 2^k - 1$, need to be precomputed, but only for those i whose binary representation is a column of EA .

14.91 Note (*ElGamal signature verification*) The signature verification equation for the ElGamal signature scheme (Algorithm 11.64) is $\alpha^{h(m)} (\alpha^{-a})^r \equiv r^s \pmod{p}$ where p is a large prime, α a generator of \mathbb{Z}_p^* , α^a is the public key, and (r, s) is a signature for message m . It would appear that three exponentiations and one multiplication are required to verify the equation. If $t = \lceil \lg p \rceil$ and Algorithm 11.64 is applied, the number of squarings is $3(t - 1)$ and the number of multiplications is, on average, $3t/2$. Hence, one would expect to perform about $(9t - 4)/2$ multiplications and squarings modulo p . Algorithm 14.88 can reduce the number of computations substantially if the verification equation is rewritten as $\alpha^{h(m)} (\alpha^{-a})^r r^{-s} \equiv 1 \pmod{p}$. Taking $g_0 = \alpha$, $g_1 = \alpha^{-a}$, $g_2 = r$, and $e_0 = h(m) \bmod (p - 1)$, $e_1 = r \bmod (p - 1)$, $e_2 = -s \bmod (p - 1)$ in Algorithm 14.88, the expected number of multiplications and squarings is $(t - 1) + (6 + (7t/8)) = (15t + 40)/8$. (For random exponents, one would expect that, on average, $\frac{7}{8}$ of the columns of EA will be non-zero and necessitate a non-trivial multiplication.) This is only about 25% more costly than a single exponentiation computed by Algorithm 14.79.

(iv) Additive notation

Algorithms 14.76 and 14.79 have been described in the setting of a multiplicative group. Algorithm 14.92 uses the methodology of Algorithm 14.79 to perform efficient multiplication in an additive group G . (For example, the group formed by the points on an elliptic curve over a finite field uses additive notation.) Multiplication in an additive group corresponds to exponentiation in a multiplicative group.

14.92 Algorithm Left-to-right binary multiplication in an additive group

INPUT: $g \in G$, where G is an additive group, and a positive integer $e = (e_t e_{t-1} \cdots e_1 e_0)_2$.
 OUTPUT: $e \cdot g$.

1. $A \leftarrow 0$.
2. For i from t down to 0 do the following:
 - 2.1 $A \leftarrow A + A$.
 - 2.2 If $e_i = 1$ then $A \leftarrow A + g$.
3. Return(A).

14.93 Note (the additive group \mathbb{Z}_m)

- (i) If G is the additive group \mathbb{Z}_m , then Algorithm 14.92 provides a method for doing modular multiplication. For example, if $a, b \in \mathbb{Z}_m$, then $a \cdot b \bmod m$ can be computed using Algorithm 14.92 by taking $g = a$ and $e = b$, provided b is written in binary.
- (ii) If $a, b \in \mathbb{Z}_m$, then $a < m$ and $b < m$. The accumulator A in Algorithm 14.92 never contains an integer as large as $2m$; hence, modular reduction of the value in the accumulator can be performed by a simple subtraction when $A \geq m$; thus no divisions are required.
- (iii) Algorithms 14.82 and 14.83 can also be used for modular multiplication. In the case of the additive group \mathbb{Z}_m , the time required to do modular multiplication can be improved at the expense of precomputing a table of residues modulo m . For a left-to-right k -ary exponentiation scheme, the table will contain $2^k - 1$ residues modulo m .

(v) Montgomery exponentiation

The introductory remarks to §14.3.2 outline an application of the Montgomery reduction method for exponentiation. Algorithm 14.94 below combines Algorithm 14.79 and Algorithm 14.36 to give a *Montgomery exponentiation algorithm* for computing $x^e \bmod m$. Note the definition of m' requires that $\gcd(m, R) = 1$. For integers u and v where $0 \leq u, v < m$, define $\text{Mont}(u, v)$ to be $uvR^{-1} \bmod m$ as computed by Algorithm 14.36.

14.94 Algorithm Montgomery exponentiation

INPUT: $m = (m_{l-1} \cdots m_0)_b$, $R = b^l$, $m' = -m^{-1} \bmod b$, $e = (e_t \cdots e_0)_2$ with $e_t = 1$, and an integer x , $1 \leq x < m$.

OUTPUT: $x^e \bmod m$.

1. $\tilde{x} \leftarrow \text{Mont}(x, R^2 \bmod m)$, $A \leftarrow R \bmod m$. ($R \bmod m$ and $R^2 \bmod m$ may be provided as inputs.)
2. For i from t down to 0 do the following:
 - 2.1 $A \leftarrow \text{Mont}(A, A)$.
 - 2.2 If $e_i = 1$ then $A \leftarrow \text{Mont}(A, \tilde{x})$.
3. $A \leftarrow \text{Mont}(A, 1)$.
4. Return(A).

14.95 Example (*Montgomery exponentiation*) Let x , m , and R be integers suitable as inputs to Algorithm 14.94. Let $e = 11 = (1011)_2$; here, $t = 3$. The following table displays the values of $A \bmod m$ at the end of each iteration of step 2, and after step 3. \square

i	3	2	1	0	Step 3
$A \bmod m$	\tilde{x}	$\tilde{x}^2 R^{-1}$	$\tilde{x}^5 R^{-4}$	$\tilde{x}^{11} R^{-10}$	$\text{Mont}(A, 1) = \tilde{x}^{11} R^{-11} = x^{11}$

14.96 Note (*computational efficiency of Montgomery exponentiation*)

- (i) Table 14.17 displays the average number of single-precision multiplications required for each step of Algorithm 14.94. The expected number of single-precision multiplications to compute $x^e \bmod m$ by Algorithm 14.94 is $3l(l+1)(t+1)$.
- (ii) Each iteration of step 2 in Algorithm 14.94 applies Algorithm 14.36 at a cost of $2l(l+1)$ single-precision multiplications but no single-precision divisions. A similar algorithm for modular exponentiation based on classical modular multiplication (Algorithm 14.28) would similarly use $2l(l+1)$ single-precision multiplications per iteration but also l single-precision divisions.
- (iii) Any of the other exponentiation algorithms discussed in §14.6.1 can be combined with Montgomery reduction to give other Montgomery exponentiation algorithms.

Step	1	2	3
Number of Montgomery multiplications	1	$\frac{3}{2}t$	1
Number of single-precision multiplications	$2l(l+1)$	$3tl(l+1)$	$l(l+1)$

Table 14.17: Average number of single-precision multiplications per step of Algorithm 14.94.

14.6.2 Fixed-exponent exponentiation algorithms

There are numerous situations in which a number of exponentiations by a fixed exponent must be performed. Examples include RSA encryption and decryption, and ElGamal decryption. This subsection describes selected algorithms which improve the repeated square-and-multiply algorithms of §14.6.1 by reducing the number of multiplications.

(i) Addition chains

The purpose of an addition chain is to minimize the number of multiplications required for an exponentiation.

14.97 Definition An *addition chain* V of length s for a positive integer e is a sequence u_0, u_1, \dots, u_s of positive integers, and an associated sequence w_1, \dots, w_s of pairs $w_i = (i_1, i_2)$, $0 \leq i_1, i_2 < i$, having the following properties:

- (i) $u_0 = 1$ and $u_s = e$; and
- (ii) for each u_i , $1 \leq i \leq s$, $u_i = u_{i_1} + u_{i_2}$.

14.98 Algorithm Addition chain exponentiation

INPUT: a group element g , an addition chain $V = (u_0, u_1, \dots, u_s)$ of length s for a positive integer e , and the associated sequence w_1, \dots, w_s , where $w_i = (i_1, i_2)$.

OUTPUT: g^e .

1. $g_0 \leftarrow g$.
2. For i from 1 to s do: $g_i \leftarrow g_{i_1} \cdot g_{i_2}$.
3. Return(g_s).

14.99 Example (*addition chain exponentiation*) An addition chain of length 5 for $e = 15$ is $u_0 = 1, u_1 = 2, u_2 = 3, u_3 = 6, u_4 = 12, u_5 = 15$. The following table displays the values of w_i and g_i during each iteration of Algorithm 14.98 for computing g^{15} . \square

i	0	1	2	3	4	5
w_i	—	(0, 0)	(0, 1)	(2, 2)	(3, 3)	(2, 4)
g_i	g	g^2	g^3	g^6	g^{12}	g^{15}

14.100 Remark (*addition chains and binary representations*) Given the binary representation of an exponent e , it is a relatively simple task to construct an addition chain directly from this representation. Chains constructed in this way generally do not provide the shortest addition chain possible for the given exponent. The methods for exponentiation described in §14.6.1 could be phrased in terms of addition chains, but this is typically not done.

14.101 Note (*computational efficiency of addition chain exponentiation*) Given an addition chain of length s for the positive integer e , Algorithm 14.98 computes g^e for any $g \in G$, $g \neq 1$, using exactly s multiplications.

14.102 Fact If l is the length of a shortest addition chain for a positive integer e , then $l \geq (\lg e + \lg \text{wt}(e) - 2.13)$, where $\text{wt}(e)$ is the number of 1's in the binary representation of e . An upper bound of $(\lceil \lg e \rceil + \text{wt}(e) - 1)$ is obtained by constructing an addition chain for e from its binary representation. Determining a shortest addition chain for e is known to be an NP-hard problem.

(ii) Vector-addition chains

Algorithms 14.88 and 14.104 are useful for computing $g_0^{e_0} g_1^{e_1} \cdots g_{k-1}^{e_{k-1}}$ where g_0, g_1, \dots, g_{k-1} are arbitrary elements in a group G and e_0, e_1, \dots, e_{k-1} are fixed positive integers. These algorithms can also be used to advantage when the exponents are not necessarily fixed values (see Note 14.91). Algorithm 14.104 makes use of vector-addition chains.

14.103 Definition Let s and k be positive integers and let v_i denote a k -dimensional vector of non-negative integers. An ordered set $V = \{v_i : -k + 1 \leq i \leq s\}$ is called a *vector-addition chain* of length s and dimension k if V satisfies the following:

- (i) Each v_i , $-k + 1 \leq i \leq 0$, has a 0 in each coordinate position, except for coordinate position $i + k - 1$, which is a 1. (Coordinate positions are labeled 0 through $k - 1$.)
- (ii) For each v_i , $1 \leq i \leq s$, there exists an associated pair of integers $w_i = (i_1, i_2)$ such that $-k + 1 \leq i_1, i_2 < i$ and $v_i = v_{i_1} + v_{i_2}$ ($i_1 = i_2$ is allowed).

Example 14.105 illustrates a sample vector-addition chain. Let $V = \{v_i : -k + 1 \leq i \leq s\}$ be a vector-addition chain of length s and dimension k with associated sequence w_1, \dots, w_s . Algorithm 14.104 computes $g_0^{e_0} g_1^{e_1} \cdots g_{k-1}^{e_{k-1}}$ where $v_s = (e_0, e_1, \dots, e_{k-1})$.

14.104 Algorithm Vector-addition chain exponentiation

INPUT: group elements g_0, g_1, \dots, g_{k-1} and a vector-addition chain V of length s and dimension k with associated sequence w_1, \dots, w_s , where $w_i = (i_1, i_2)$.

OUTPUT: $g_0^{e_0} g_1^{e_1} \cdots g_{k-1}^{e_{k-1}}$ where $v_s = (e_0, e_1, \dots, e_{k-1})$.

1. For i from $(-k + 1)$ to 0 do: $a_i \leftarrow g_{i+k-1}$.
2. For i from 1 to s do: $a_i \leftarrow a_{i_1} \cdot a_{i_2}$.
3. Return(a_s).

14.105 Example (*vector-addition chain exponentiation*) A vector-addition chain V of length $s = 9$ and dimension $k = 3$ is displayed in the following table.

v_{-2}	v_{-1}	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
1	0	0	1	2	2	3	5	6	12	15	30
0	1	0	0	0	1	1	2	2	4	5	10
0	0	1	1	2	2	2	4	5	10	12	24

The following table displays the values of w_i and a_i during each iteration of step 2 in Algorithm 14.104 for computing $g_0^{30} g_1^{10} g_2^{24}$. Nine multiplications are required. \square

i	1	2	3	4	5	6	7	8	9
w_i	$(-2, 0)$	$(1, 1)$	$(-1, 2)$	$(-2, 3)$	$(3, 4)$	$(1, 5)$	$(6, 6)$	$(4, 7)$	$(8, 8)$
a_i	$g_0 g_2$	$g_0^2 g_2^2$	$g_0^2 g_1 g_2^2$	$g_0^3 g_1 g_2^2$	$g_0^5 g_1^2 g_2^4$	$g_0^6 g_1^2 g_2^5$	$g_0^{12} g_1^4 g_2^{10}$	$g_0^{15} g_1^5 g_2^{12}$	$g_0^{30} g_1^{10} g_2^{24}$

14.106 Note (*computational efficiency of vector-addition chain exponentiation*)

- (i) (*multiplications*) Algorithm 14.104 performs exactly s multiplications for a vector-addition chain of length s . To compute $g_0^{e_0} g_1^{e_1} \cdots g_{k-1}^{e_{k-1}}$ using Algorithm 14.104, one would like to find a vector-addition chain of length s and dimension k with $v_s = (e_0, e_1, \dots, e_{k-1})$, where s is as small as possible (see Fact 14.107).

- (ii) (*storage*) Algorithm 14.104 requires intermediate storage for the elements a_i , $-k + 1 \leq i < t$, at the t^{th} iteration of step 2. If not all of these are required for succeeding iterations, then they need not be stored. Algorithm 14.88 provides a special case of Algorithm 14.104 where the intermediate storage is no larger than $2^k - 1$ vectors of dimension k .

14.107 Fact The minimum value of s in Note 14.106(i) satisfies the following bound, where $M = \max\{e_i : 0 \leq i \leq k-1\}$ and c is a constant:

$$s \leq k - 1 + \lg M + ck \cdot \lg M / \lg \lg(M + 2).$$

14.108 Example (*vector-addition chains from binary representations*) The vector-addition chain implicit in Algorithm 14.88 is not necessarily of minimum length. The vector-addition chain associated with Example 14.89 is displayed in Table 14.18. This chain is longer than the one used in Example 14.105. The advantage of Algorithm 14.88 is that the vector-addition chain does not have to be explicitly provided to the algorithm. In view of this, Algorithm 14.88 can be applied more generally to situations where the exponents are not necessarily fixed. \square

v_{-2}	v_{-1}	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
1	0	0	1	1	1	2	3	6	7	14	15	30
0	1	0	1	1	0	0	1	2	2	4	5	10
0	0	1	0	1	1	2	3	6	6	12	12	24

Table 14.18: Binary vector-addition chain exponentiation (see Example 14.108).

14.6.3 Fixed-base exponentiation algorithms

Three methods are presented for exponentiation when the base g is fixed and the exponent e varies. With a fixed base, precomputation can be done once and used for many exponentiations. For example, Diffie-Hellman key agreement (Protocol 12.47) requires the computation of α^x , where α is a fixed element in \mathbb{Z}_p^* .

For each of the algorithms described in this section, $\{b_0, b_1, \dots, b_t\}$ is a set of integers for some $t \geq 0$, such that any exponent $e \geq 1$ (suitably bounded) can be written as $e = \sum_{i=0}^t e_i b_i$, where $0 \leq e_i < h$ for some fixed positive integer h . For example, if e is any $(t+1)$ -digit base b integer with $b \geq 2$, then $b_i = b^i$ and $h = b$ are possible choices.

Algorithms 14.109 and 14.113 are two fixed-base exponentiation methods. Both require precomputation of the exponentials $g^{b_0}, g^{b_1}, \dots, g^{b_t}$, e.g., using one of the algorithms from §14.6.1. The precomputation needed for Algorithm 14.117 is more involved and is explicitly described in Algorithm 14.116.

(i) Fixed-base windowing method

Algorithm 14.109 takes as input the precomputed exponentials $g_i = g^{b_i}$, $0 \leq i \leq t$, and positive integers h and $e = \sum_{i=0}^t e_i b_i$ where $0 \leq e_i < h$, $0 \leq i \leq t$. The basis for the algorithm is the observation that $g^e = \prod_{i=0}^t g_i^{e_i} = \prod_{j=1}^{h-1} (\prod_{e_i=j} g_i)^j$.

14.109 Algorithm Fixed-base windowing method for exponentiation

INPUT: $\{g^{b_0}, g^{b_1}, \dots, g^{b_t}\}$, $e = \sum_{i=0}^t e_i b_i$, and h .
 OUTPUT: g^e .

1. $A \leftarrow 1$, $B \leftarrow 1$.
2. For j from $(h - 1)$ down to 1 do the following:
 - 2.1 For each i for which $e_i = j$ do: $B \leftarrow B \cdot g^{b_i}$.
 - 2.2 $A \leftarrow A \cdot B$.
3. Return(A).

14.110 Example (*fixed-base windowing exponentiation*) Precompute the group elements $g^1, g^4, g^{16}, g^{64}, g^{256}$. To compute g^e for $e = 862 = (31132)_4$, take $t = 4$, $h = 4$, and $b_i = 4^i$ for $0 \leq i \leq 4$, in Algorithm 14.109. The following table displays the values of A and B at the end of each iteration of step 2. \square

j	—	3	2	1
B	1	$g^4 g^{256} = g^{260}$	$g^{260} g = g^{261}$	$g^{261} g^{16} g^{64} = g^{341}$
A	1	g^{260}	$g^{260} g^{261} = g^{521}$	$g^{521} g^{341} = g^{862}$

14.111 Note (*computational efficiency of fixed-base windowing exponentiation*)

- (i) (*number of multiplications*) Suppose $t + h \geq 2$. Only multiplications where both operands are distinct from 1 are counted. Step 2.2 is executed $h - 1$ times, but at least one of these multiplications involves an operand with value 1 (A is initialized to 1). Since B is also initially 1, at most t multiplications are done in step 2.1. Thus, Algorithm 14.109 computes g^e with at most $t + h - 2$ multiplications (cf. Note 14.112).
- (ii) (*storage*) Storage is required for the $t + 1$ group elements g_i , $0 \leq i \leq t$.

14.112 Note (*a particular case*) The most obvious application of Algorithm 14.109 is the case where the exponent e is represented in radix b . If $e = \sum_{i=0}^t e_i b^i$, then $g_i = g^{b^i}$, $0 \leq i \leq t$, are precomputed. If e is randomly selected from $\{0, 1, \dots, m - 1\}$, then $t + 1 \leq \lceil \log_b m \rceil$ and, on average, $\frac{1}{b}$ of the base b digits in e will be 0. In this case, the expected number of multiplications is $\frac{b-1}{b} \lceil \log_b m \rceil + b - 3$. If m is a 512-bit integer and $b = 32$, then 128.8 multiplications are needed on average, 132 in the worst case; 103 values must be stored.

(ii) Fixed-base Euclidean method

Let $\{x_0, x_1, \dots, x_t\}$ be a set of integers with $t \geq 2$. Define M to be an integer in the interval $[0, t]$ such that $x_M \geq x_i$ for all $0 \leq i \leq t$. Define N to be an integer in the interval $[0, t]$, $N \neq M$, such that $e_N \geq e_i$ for all $0 \leq i \leq t$, $i \neq M$.

14.113 Algorithm Fixed-base Euclidean method for exponentiation

INPUT: $\{g^{b_0}, g^{b_1}, \dots, g^{b_t}\}$ and $e = \sum_{i=0}^t e_i b_i$.
 OUTPUT: g^e .

1. For i from 0 to t do the following: $g_i \leftarrow g^{b_i}$, $x_i \leftarrow e_i$.
2. Determine the indices M and N for $\{x_0, x_1, \dots, x_t\}$.
3. While $x_N \neq 0$ do the following:
 - 3.1 $q \leftarrow \lfloor x_M / x_N \rfloor$, $g_N \leftarrow (g_M)^q \cdot g_N$, $x_M \leftarrow x_M \bmod x_N$.

3.2 Determine the indices M and N for $\{x_0, x_1, \dots, x_t\}$.

4. Return($g_M^{x_M}$).

14.114 Example (*fixed-base Euclidean method*) This example repeats the computation of g^e , $e = 862$ done in Example 14.110, but now uses Algorithm 14.113. Take $b_0 = 1$, $b_1 = 16$, $b_2 = 256$. Then $e = (3, 5, 14)_{16}$. Precompute g^1, g^{16}, g^{256} . Table 14.19 illustrates the steps performed by Algorithm 14.113. Notice that for this example, Algorithm 14.113 does 8

x_0	x_1	x_2	M	N	q	g_0	g_1	g_2
14	5	3	0	1	2	g	g^{18}	g^{256}
4	5	3	1	0	1	g^{19}	g^{18}	g^{256}
4	1	3	0	2	1	g^{19}	g^{18}	g^{275}
1	1	3	2	1	3	g^{19}	g^{843}	g^{275}
1	1	0	0	1	1	g^{19}	g^{862}	g^{275}
0	1	0	1	0	—	g^{19}	g^{602}	g^{275}

Table 14.19: Fixed-base Euclidean method to compute g^{862} (see Example 14.114).

multiplications, whereas Algorithm 14.109 needs only 6 to do the same computation. Storage requirements for Algorithm 14.113 are, however, smaller. The vector-addition chain (Definition 14.103) corresponding to this example is displayed in the following table. □

v_{-2}	v_{-1}	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
1	0	0	2	2	3	3	6	9	11	14
0	1	0	0	1	1	1	2	3	4	5
0	0	1	0	0	0	1	2	3	3	3

14.115 Note (*fixed-base Euclidean vs. fixed-base windowing methods*)

- (i) In most cases, the quotient q computed in step 3.1 of Algorithm 14.113 is 1. For a given base b , the computational requirements of this algorithm are not significantly greater than those of Algorithm 14.109.
- (ii) Since the division algorithm is logarithmic in the size of the inputs, Algorithm 14.113 can take advantage of a larger value of h than Algorithm 14.109. This results in less storage for precomputed values.

(iii) Fixed-base comb method

Algorithm 14.117 computes g^e where $e = (e_t e_{t-1} \dots e_1 e_0)_2$, $t \geq 1$. Select an integer h , $1 \leq h \leq t+1$ and compute $a = \lceil (t+1)/h \rceil$. Select an integer v , $1 \leq v \leq a$, and compute $b = \lceil a/v \rceil$. Clearly, $ah \geq t+1$. Let $X = R_{h-1} || R_{h-2} || \dots || R_0$ be a bitstring formed from e by padding (if necessary) e on the left with 0's, so that X has bitlength ah and each R_i , $0 \leq i \leq h-1$, is a bitstring of length a . Form an $h \times a$ array EA (called the *exponent array*) where row i of EA is the bitstring R_i , $0 \leq i \leq h-1$. Algorithm 14.116 is the precomputation required for Algorithm 14.117.

14.116 Algorithm Precomputation for Algorithm 14.117

INPUT: group element g and parameters h, v, a , and b (defined above).

OUTPUT: $\{G[j][i] : 1 \leq i < 2^h, 0 \leq j < v\}$.

1. For i from 0 to $(h-1)$ do: $g_i \leftarrow g^{2^{ia}}$.
2. For i from 1 to (2^h-1) (where $i = (i_{h-1} \cdots i_0)_2$), do the following:
 - 2.1 $G[0][i] \leftarrow \prod_{j=0}^{h-1} g_j^{i_j}$.
 - 2.2 For j from 1 to $(v-1)$ do: $G[j][i] \leftarrow (G[0][i])^{2^{jb}}$.
3. Return($\{G[j][i] : 1 \leq i < 2^h, 0 \leq j < v\}$).

Let $I_{j,k}, 0 \leq k < b, 0 \leq j < v$, be the integer whose binary representation is column $(jb+k)$ of EA , where column 0 is on the right and the least significant bits of a column are at the top.

14.117 Algorithm Fixed-base comb method for exponentiation

INPUT: g, e and $\{G[j][i] : 1 \leq i < 2^h, 0 \leq j < v\}$ (precomputed in Algorithm 14.116).

OUTPUT: g^e .

1. $A \leftarrow 1$.
2. For k from $(b-1)$ down to 0 do the following:
 - 2.1 $A \leftarrow A \cdot A$.
 - 2.2 For j from $(v-1)$ down to 0 do: $A \leftarrow G[j][I_{j,k}] \cdot A$.
3. Return(A).

14.118 Example (*fixed-base comb method for exponentiation*) Let $t = 9$ and $h = 3$; then $a = \lceil 10/3 \rceil = 4$. Let $v = 2$; then $b = \lceil a/v \rceil = 2$. Suppose the exponent input to Algorithm 14.117 is $e = (e_9 e_8 \cdots e_1 e_0)_2$. Form the bitstring $X = x_{11} x_{10} \cdots x_1 x_0$ where $x_i = e_i, 0 \leq i \leq 9$, and $x_{11} = x_{10} = 0$. The following table displays the exponent array EA .

$I_{1,1}$	$I_{1,0}$	$I_{0,1}$	$I_{0,0}$
x_3	x_2	x_1	x_0
x_7	x_6	x_5	x_4
x_{11}	x_{10}	x_9	x_8

The precomputed values from Algorithm 14.116 are displayed below. Recall that $g_i = g^{2^{ia}}, 0 \leq i < 3$.

i	1	2	3	4	5	6	7
$G[0][i]$	g_0	g_1	$g_1 g_0$	g_2	$g_2 g_0$	$g_2 g_1$	$g_2 g_1 g_0$
$G[1][i]$	g_0^4	g_1^4	$g_1^4 g_0^4$	g_2^4	$g_2^4 g_0^4$	$g_2^4 g_1^4$	$g_2^4 g_1^4 g_0^4$

Finally, the following table displays the steps in Algorithm 14.117 for EA .

		$A = g_0^{l_0} g_1^{l_1} g_2^{l_2}$		
k	j	l_0	l_1	l_2
1	—	0	0	0
1	1	$4x_3$	$4x_7$	$4x_{11}$
1	0	$4x_3 + x_1$	$4x_7 + x_5$	$4x_{11} + x_9$
0	—	$8x_3 + 2x_1$	$8x_7 + 2x_5$	$8x_{11} + 2x_9$
0	1	$8x_3 + 2x_1 + 4x_2$	$8x_7 + 2x_5 + 4x_6$	$8x_{11} + 2x_9 + 4x_{10}$
0	0	$8x_3 + 2x_1 + 4x_2 + x_0$	$8x_7 + 2x_5 + 4x_6 + x_4$	$8x_{11} + 2x_9 + 4x_{10} + x_8$

The last row of the table corresponds to $g^{\sum_{i=0}^{11} x_i 2^i} = g^e$. □

14.119 Note (*computational efficiency of fixed-base comb method*)

- (i) (*number of multiplications*) Algorithm 14.117 requires at most one multiplication for each column of EA . The right-most column of EA requires a multiplication with the initial value 1 of the accumulator A . The algorithm also requires a squaring of the accumulator A for each k , $0 \leq k < b$, except for $k = b - 1$ when A has value 1. Discounting multiplications by 1, the total number of non-trivial multiplications (including squarings) is, at most, $a + b - 2$.
- (ii) (*storage*) Algorithm 14.117 requires storage for the $v(2^h - 1)$ precomputed group elements (Algorithm 14.116). If squaring is a relatively simple operation compared to multiplication in the group, then some space-saving can be achieved by storing only $2^h - 1$ group elements (i.e., only those elements computed in step 2.1 of Algorithm 14.116).
- (iii) (*trade-offs*) Since h and v are independent of the number of bits in the exponent, selection of these parameters can be made based on the amount of storage available vs. the amount of time (determined by multiplication) to do the computation.

14.7 Exponent recoding

Another approach to reducing the number of multiplications in the basic repeated square-and-multiply algorithms (§14.6.1) is to replace the binary representation of the exponent e with a representation which has fewer non-zero terms. Since the binary representation is unique (Fact 14.1), finding a representation with fewer non-zero components necessitates the use of digits besides 0 and 1. Transforming an exponent from one representation to another is called *exponent recoding*. Many techniques for exponent recoding have been proposed in the literature. This section describes two possibilities: signed-digit representation (§14.7.1) and string-replacement representation (§14.7.2).

14.7.1 Signed-digit representation

14.120 Definition If $e = \sum_{i=0}^t d_i 2^i$ where $d_i \in \{0, 1, -1\}$, $0 \leq i \leq t$, then $(d_t \cdots d_1 d_0)_{SD}$ is called a *signed-digit representation with radix 2* for the integer e .

Unlike the binary representation, the signed-digit representation of an integer is not unique. The binary representation is an example of a signed-digit representation. Let e be a positive integer whose binary representation is $(e_{t+1}e_te_{t-1} \cdots e_1e_0)_2$, with $e_{t+1} = e_t = 0$. Algorithm 14.121 constructs a signed-digit representation for e having at most $t + 1$ digits and the smallest possible number of non-zero terms.

14.121 Algorithm Signed-digit exponent recoding

INPUT: a positive integer $e = (e_{t+1}e_te_{t-1} \cdots e_1e_0)_2$ with $e_{t+1} = e_t = 0$.

OUTPUT: a signed-digit representation $(d_t \cdots d_1d_0)_{SD}$ for e . (See Definition 14.120.)

1. $c_0 \leftarrow 0$.
2. For i from 0 to t do the following:
 - 2.1 $c_{i+1} \leftarrow \lfloor (e_i + e_{i+1} + c_i)/2 \rfloor$, $d_i \leftarrow e_i + c_i - 2c_{i+1}$.
3. Return $((d_t \cdots d_1d_0)_{SD})$.

14.122 Example (*signed-digit exponent recoding*) Table 14.20 lists all possible inputs to the i^{th} iteration of step 2, and the corresponding outputs. If $e = (1101110111)_2$, then Algorithm 14.121 produces the signed-digit representation $e = (100\bar{1}000\bar{1}00\bar{1})_{SD}$ where $\bar{1} = -1$. Note that $e = 2^9 + 2^8 + 2^6 + 2^5 + 2^4 + 2^2 + 2 + 1 = 2^{10} - 2^7 - 2^3 - 1$. \square

inputs	e_i	0	0	0	0	1	1	1	1
	c_i	0	0	1	1	0	0	1	1
	e_{i+1}	0	1	0	1	0	1	0	1
outputs	c_{i+1}	0	0	0	1	0	1	1	1
	d_i	0	0	1	-1	1	-1	0	0

Table 14.20: Signed-digit exponent recoding (see Example 14.122).

14.123 Definition A signed-digit representation of an integer e is said to be *sparse* if no two non-zero entries are adjacent in the representation.

14.124 Fact (*sparse signed-digit representation*)

- (i) Every integer e has a unique sparse signed-digit representation.
- (ii) A sparse signed-digit representation for e has the smallest number of non-zero entries among all signed-digit representations for e .
- (iii) The signed-digit representation produced by Algorithm 14.121 is sparse.

14.125 Note (*computational efficiency of signed-digit exponent recoding*)

- (i) Signed-digit exponent recoding as per Algorithm 14.121 is very efficient, and can be done by table look-up (using Table 14.20).
- (ii) When e is given in a signed-digit representation, computing g^e requires both g and g^{-1} . If g is a fixed base, then g^{-1} can be precomputed. For a variable base g , unless g^{-1} can be computed very quickly, recoding an exponent to signed-digit representation may not be worthwhile.

14.7.2 String-replacement representation

14.126 Definition Let $k \geq 1$ be a positive integer. A non-negative integer e is said to have a k -ary string-replacement representation $(f_{t-1}f_{t-2} \cdots f_1f_0)_{SR(k)}$, denoted $SR(k)$, if $e = \sum_{i=0}^{t-1} f_i 2^i$ and $f_i \in \{2^j - 1 : 0 \leq j \leq k\}$ for $0 \leq i \leq t-1$.

14.127 Example (*non-uniqueness of string-replacement representations*) A string-replacement representation for a non-negative integer is generally not unique. The binary representation is a 1-ary string-replacement representation. If $k = 3$ and $e = 987 = (1111011011)_2$, then some other string-replacements of e are $(303003003)_{SR(3)}$, $(1007003003)_{SR(3)}$, and $(71003003)_{SR(3)}$. \square

14.128 Algorithm k -ary string-replacement representation

INPUT: $e = (e_{t-1}e_{t-2} \cdots e_1e_0)_2$ and positive integer $k \geq 2$.

OUTPUT: $e = (f_{t-1}f_{t-2} \cdots f_1f_0)_{SR(k)}$.

1. For i from k down to 2 do the following: starting with the most significant digit of $e = (e_{t-1}e_{t-2} \cdots e_1e_0)_2$, replace each consecutive string of i ones with a string of length i consisting of $i - 1$ zeros in the high-order string positions and the integer $2^i - 1$ in the low-order position.
 2. Return $((f_{t-1}f_{t-2} \cdots f_1f_0)_{SR(k)})$.
-

14.129 Example (k -ary string-replacement) Suppose $e = (110111110011101)_2$ and $k = 3$. The $SR(3)$ representations of e at the end of each of the two iterations of Algorithm 14.128 are $(110007110000701)_{SR(3)}$ and $(030007030000701)_{SR(3)}$. \square

14.130 Algorithm Exponentiation using an $SR(k)$ representation

INPUT: an integer $k \geq 2$, an element $g \in G$, and $e = (f_{t-1}f_{t-2} \cdots f_1f_0)_{SR(k)}$.

OUTPUT: g^e .

1. *Precomputation.* Set $g_1 \leftarrow g$. For i from 2 to k do: $g_{2^i-1} \leftarrow (g_{2^{i-1}-1})^2 \cdot g$.
 2. $A \leftarrow 1$.
 3. For i from $(t - 1)$ down to 0 do the following:
 - 3.1 $A \leftarrow A \cdot A$.
 - 3.2 If $f_i \neq 0$ then $A \leftarrow A \cdot g_{f_i}$.
 4. Return (A) .
-

14.131 Example ($SR(k)$ vs. left-to-right binary exponentiation) Let $e = 987 = (1111011011)_2$ and consider the 3-ary string-replacement representation $(0071003003)_{SR(3)}$. Computing g^e using Algorithm 14.79 requires 9 squarings and 7 multiplications. Algorithm 14.130 requires 2 squarings and 2 multiplications for computing g^3 and g^7 , and then 7 squarings and 3 multiplications for the main part of the algorithm. In total, the $SR(3)$ for e computes g^e with 9 squarings and 5 multiplications. \square

14.132 Note (*computational efficiency of Algorithm 14.130*) The precomputation requires $k - 1$ squarings and $k - 1$ multiplications. Algorithm 14.128 is not guaranteed to produce an $SR(k)$ representation with a minimum number of non-zero entries, but in practice it seems to give representations which are close to minimal. Heuristic arguments indicate that a randomly selected t -bit exponent will be encoded with a suitably chosen value of k to an $SR(k)$ representation having about $t/4$ non-zero entries; hence, one expects to perform $t - 1$ squarings in step 3, and about $t/4$ multiplications.

14.8 Notes and further references

§14.1

This chapter deals almost exclusively with methods to perform operations in the integers and the integers modulo some positive integer. When p is a prime number, \mathbb{Z}_p is called a finite field (Fact 2.184). There are other finite fields which have significance in cryptography. Of particular importance are those of characteristic two, \mathbb{F}_{2^m} . Perhaps the most useful property of these structures is that squaring is a *linear operator* (i.e., if $\alpha, \beta \in \mathbb{F}_{2^m}$, then $(\alpha + \beta)^2 = \alpha^2 + \beta^2$). This property leads to efficient methods for exponentiation and for inversion. Characteristic two finite fields have been used extensively in connection with error-correcting codes; for example, see Berlekamp [118] and Lin and Costello [769]. For error-correcting codes, m is typically quite small (e.g., $1 \leq m \leq 16$); for cryptographic applications, m is usually much larger (e.g., $m \geq 100$).

The majority of the algorithms presented in this chapter are best suited to software implementations. There is a vast literature on methods to perform modular multiplication and other operations in hardware. The basis for most hardware implementations for modular multiplication is efficient methods for integer addition. In particular, *carry-save adders* and *delayed-carry adders* are at the heart of the best methods to perform modular multiplication. The concept of a delayed-carry adder was proposed by Norris and Simmons [933] to produce a hardware modular multiplier which computes the product of two t -bit operands modulo a t -bit modulus in $2t$ clock cycles. Brickell [199] improved the idea to produce a modular multiplier requiring only $t + 7$ clock cycles. Enhancements of Brickell's method were given by Walter [1230]. Koç [699] gives a comprehensive survey of hardware methods for modular multiplication.

§14.2

For a treatment of radix representations including *mixed-radix representations*, see Knuth [692]. Knuth describes efficient methods for performing radix conversions. Representing and manipulating negative numbers is an important topic; for an introduction, consult the book by Koren [706].

The techniques described in §14.2 are commonly referred to as the *classical algorithms* for multiple-precision addition, subtraction, multiplication, and division. These algorithms are the most useful for integers of the size used for cryptographic purposes. For much larger integers (on the order of thousands of decimal digits), more efficient methods exist. Although not of current practical interest, some of these may become more useful as security requirements force practitioners to increase parameter sizes. A brief description of one of these methods follows.

The classical algorithm for multiplication (Algorithm 14.12) takes $O(n^2)$ bit operations for multiplying two n -bit integers. A recursive algorithm due to Karatsuba and Ofman [661] reduces the complexity of multiplying two n -bit integers to $O(n^{1.58})$. This *divide-and-conquer* method is based on the following simple observation. Suppose that x and y are n -bit integers and $n = 2t$. Then $x = 2^t x_1 + x_0$ and $y = 2^t y_1 + y_0$, where x_1, y_1 are the t high-order bits of x and y , respectively, and x_0, y_0 are the t low-order bits. Furthermore, $x \cdot y = u_2 2^{2t} + u_1 2^t + u_0$ where $u_0 = x_0 \cdot y_0$, $u_2 = x_1 \cdot y_1$ and $u_1 = (x_0 + x_1) \cdot (y_0 + y_1) - u_0 - u_2$. It follows that $x \cdot y$ can be computed by performing three multiplications of t -bit integers (as opposed to one multiplication with $2t$ -bit integers) along with two additions and two subtractions. For large values of t , the cost of the additions and subtractions is insignificant relative to the cost of the multiplications. With appropriate modifications, u_0, u_1 and

$(x_0 + x_1) \cdot (y_0 + y_1)$ can each be computed similarly. This procedure is continued on the intermediate values until the size of the integers reaches the word size of the computing device, and multiplication can be efficiently accomplished. Due to the recursive nature of the algorithm, a number of intermediate results must be stored which can add significant overhead, and detract from the algorithm's efficiency for relatively small integers. Combining the Karatsuba-Ofman method with classical multiplication may have some practical significance. For a more detailed treatment of the Karatsuba-Ofman algorithm, see Knuth [692], Koç [698], and Geddes, Czapor, and Labahn [445].

Another commonly used method for multiple-precision integer multiplication is the *discrete Fourier transform* (DFT). Although mathematically elegant and asymptotically better than the classical algorithm, it does not appear to be superior for the size of integers of practical importance to cryptography. Lipson [770] provides a well-motivated and easily readable treatment of this method.

The identity given in Note 14.18 was known to Karatsuba and Ofman [661].

§14.3

There is an extensive literature on methods for multiple-precision modular arithmetic. A detailed treatment of methods for performing modular multiplication can be found in Knuth [692]. Koç [698] and Bosselaers, Govaerts, and Vandewalle [176] provide comprehensive but brief descriptions of the classical method for modular multiplication.

Montgomery reduction (Algorithm 14.32) is due to Montgomery [893], and is one of the most widely used methods in practice for performing modular exponentiation (Algorithm 14.94). Dussé and Kaliski [361] discuss variants of Montgomery's method. Montgomery reduction is a generalization of a much older technique due to Hensel (see Shand and Vuillemin [1119] and Bosselaers, Govaerts, and Vandewalle [176]). Hensel's observation is the following. If m is an odd positive integer less than 2^k (k a positive integer) and T is some integer such that $2^k \leq T < 2^{k+1}$, then $R_0 = (T + q_0 m)/2$, where $q_0 = T \bmod 2$ is an integer and $R_0 \equiv T2^{-1} \bmod m$. More generally, $R_i = (R_{i-1} + q_i m)/2$, where $q_i = R_{i-1} \bmod 2$ is an integer and $R_i \equiv N2^{-i+1} \bmod m$. Since $T < 2^{2k}$, it follows that $R_{k-1} < 2m$.

Barrett reduction (Algorithm 14.42) is due to Barrett [75]. Bosselaers, Govaerts, and Vandewalle [176] provide a clear and concise description of the algorithm along with motivation and justification for various choices of parameters and steps, and compare three alternative methods: classical (§14.3.1), Montgomery reduction (§14.3.2), and Barrett reduction (§14.3.3). This comparison indicates that there is not a significant difference in performance between the three methods, provided the precomputation necessary for Montgomery and Barrett reduction is ignored. Montgomery exponentiation is shown to be somewhat better than the other two methods. The conclusions are based on both theoretical analysis and machine implementation for various sized moduli. Koç, Acar, and Kaliski [700] provide a more detailed comparison of various Montgomery multiplication algorithms; see also Naccache, M'Raihi, and Raphaëli [915]. Naccache and M'silti [917] provide proofs for the correctness of Barrett reduction along with a possible optimization.

Mohan and Adiga [890] describe a special case of Algorithm 14.47 where $b = 2$.

Hong, Oh, and Yoon [561] proposed new methods for modular multiplication and modular squaring. They report improvements of 50% and 30%, respectively, on execution times over Montgomery's method for multiplication and squaring. Their approach to modular multiplication interleaves multiplication and modular reduction and uses precomputed tables such that one operand is always single-precision. Squaring uses recursion and pre-

computed tables and, unlike Montgomery's method, also integrates the multiplication and reduction steps.

§14.4

The binary gcd algorithm (Algorithm 14.54) is due to Stein [1170]. An analysis of the algorithm is given by Knuth [692]. Harris [542] proposed an algorithm for computing gcd's which combines the classical Euclidean algorithm (Algorithm 2.104) and binary operations; the method is called the *binary Euclidean algorithm*.

Lehmer's gcd algorithm (Algorithm 14.57), due to Lehmer [743], determines the gcd of two positive multiple-precision integers using mostly single-precision operations. This has the advantage of using the hardware divide in the machine and only periodically resorting to an algorithm such as Algorithm 14.20 for a multiple-precision divide. Knuth [692] gives a comprehensive description of the algorithm along with motivation of its correctness. Cohen [263] provides a similar discussion, but without motivation. Lehmer's gcd algorithm is readily adapted to the extended Euclidean algorithm (Algorithm 2.107).

According to Sorenson [1164], the binary gcd algorithm is the most efficient method for computing the greatest common divisor. Jebelean [633] suggests that Lehmer's gcd algorithm is more efficient. Sorenson [1164] also describes a k -ary version of the binary gcd algorithm, and proves a worst-case running time of $O(n^2 / \lg n)$ bit operations for computing the gcd of two n -bit integers.

The binary extended gcd algorithm was first described by Knuth [692], who attributes it to Penk. Algorithm 14.61 is due to Bach and Shallit [70], who also give a comprehensive and clear analysis of several gcd and extended gcd algorithms. Norton [934] described a version of the binary extended gcd algorithm which is somewhat more complicated than Algorithm 14.61. Gordon [516] proposed a method for computing modular inverses, derived from the classical extended Euclidean algorithm (Algorithm 2.107) with multiple-precision division replaced by an approximation to the quotient by an appropriate power of 2; no analysis of the expected running time is given, but observed results on moduli of specific sizes are described.

The *Montgomery inverse* of $a \bmod m$ is defined to be $a^{-1}2^t \bmod m$ where t is the bitlength of m . Kaliski [653] extended ideas of Guyot [534] on the right-shift binary extended Euclidean algorithm, and presented an algorithm for computing the Montgomery inverse.

§14.5

Let m_i , $1 \leq i \leq t$, be a set of pairwise relatively prime positive integers which define a residue number system (RNS). If $n = \prod_{i=1}^t m_i$ then this RNS provides an effective method for computing the product of integers modulo n where the integers and the product are represented in the RNS. If n is a positive integer where the m_i do not necessarily divide n , then a method for performing arithmetic modulo n entirely within the RNS is not obvious. Couveignes [284] and Montgomery and Silverman [895] propose an interesting method for accomplishing this. Further research in the area is required to determine if this approach is competitive with or better than the modular multiplication methods described in §14.3.

Algorithm 14.71 is due to Garner [443]. A detailed discussion of this algorithm and variants of it are given by Knuth [692]; see also Cohen [263]. Algorithm 2.121 for applying the Chinese remainder theorem is due to Gauss; see Bach and Shallit [70]. Gauss's algorithm is a special case of the following result due to Nagasaka, Shiue, and Ho [918]. The solution to the system of linear congruences $x \equiv a_i \pmod{m_i}$, $1 \leq i \leq t$, for pairwise relative prime moduli m_i , is equivalent to the solution to the single linear congruence $(\sum_{i=1}^t b_i M_i)x \equiv \sum_{i=1}^t a_i b_i M_i \pmod{M}$ where $M = \prod_{i=1}^t m_i$, $M_i = M/m_i$

for $1 \leq i \leq t$, for any choice of integers b_i where $\gcd(b_i, M_i) = 1$. Notice that if $\sum_{i=1}^t b_i M_i \equiv 1 \pmod{M}$, then $b_i \equiv M_i^{-1} \pmod{m_i}$, giving the special case discussed in Algorithm 2.121. Quisquater and Couvreur [1016] were the first to apply the Chinese remainder theorem to RSA decryption and signature generation.

§14.6

Knuth [692] and Bach and Shallit [70] describe the right-to-left binary exponentiation method (Algorithm 14.76). Cohen [263] provides a more comprehensive treatment of the right-to-left and left-to-right (Algorithm 14.79) binary methods along with their generalizations to the k -ary method. Koç [698] discusses these algorithms in the context of the RSA public-key cryptosystem. Algorithm 14.92 is the basis for Blakley's modular multiplication algorithm (see Blakley [149] and Koç [698]). The generalization of Blakley's method to process more than one bit per iteration (Note 14.93(iii)) is due to Quisquater and Couvreur [1016]. Quisquater and Couvreur describe an algorithm for modular exponentiation which makes use of the generalization and precomputed tables to accelerate multiplication in \mathbb{Z}_m .

For a comprehensive and detailed discussion of addition chains, see Knuth [692], where various methods for constructing addition chains (such as the *power tree* and *factor* methods) are described. Computing the shortest addition chain for a positive integer was shown to be an NP-hard problem by Downey, Leong, and Sethi [360]. The lower bound on the length of a shortest addition chain (Fact 14.102) was proven by Schönhage [1101].

An *addition sequence* for positive integers $a_1 < a_2 < \dots < a_k$ is an addition chain for a_k in which a_1, a_2, \dots, a_{k-1} appear. Yao [1257] proved that there exists an addition sequence for $a_1 < a_2 < \dots < a_k$ of length less than $\lg a_k + ck \cdot \lg a_k / \lg \lg(a_k + 2)$ for some constant c . Olivos [955] established a 1-1 correspondence between addition sequences of length l for $a_1 < a_2 < \dots < a_k$ and vector-addition chains of length $l + k - 1$ where $v_{l+k-1} = (a_1, a_2, \dots, a_k)$. These results are the basis for the inequality given in Fact 14.107. Bos and Coster [173] described a heuristic method for computing vector-addition chains. The special case of Algorithm 14.104 (Algorithm 14.88) is attributed by ElGamal [368] to Shamir.

The fixed-base windowing method (Algorithm 14.109) for exponentiation is due to Brickell et al. [204], who describe a number of variants of the basic algorithm. For b a positive integer, let S be a set of integers with the property that any integer can be expressed in base b using only coefficients from S . S is called a *basic digit set* for the base b . Brickell et al. show how basic digit sets can be used to reduce the amount of work in Algorithm 14.109 without large increases in storage requirements. De Rooij [316] proposed the fixed-base Euclidean method (Algorithm 14.113) for exponentiation; compares this algorithm to Algorithm 14.109; and provides a table of values for numbers of practical importance. The fixed-base comb method (Algorithm 14.117) for exponentiation is due to Lim and Lee [767]. For a given exponent size, they discuss various possibilities for the choice of parameters h and v , along with a comparison of their method to fixed-base windowing.

§14.7

The signed-digit exponent recoding algorithm (Algorithm 14.121) is due to Reitwiesner [1031]. A simpler description of the algorithm was given by Hwang [566]. Booth [171] described another algorithm for producing a signed-digit representation, but not necessarily one with the minimum possible non-zero components. It was originally given in terms of the additive group of integers where exponentiation is referred to as multiplication. In this case, $-g$ is easily computed from g . The additive abelian group formed from the points on an elliptic curve over a finite field is another example where signed-digit representation is very useful (see Morain and Olivos [904]). Zhang [1267] described a modified signed-digit

representation which requires on average $t/3$ multiplications for a square-and-multiply algorithm for t -bit exponents. A slightly more general version of Algorithm 14.121, given by Jedwab and Mitchell [634], does not require as input a binary representation of the exponent e but simply a signed-digit representation. For binary inputs, the algorithms of Reitwiesner and Jedwab-Mitchell are the same. Fact 14.124 is due to Jedwab and Mitchell [634].

String-replacement representations were introduced by Gollmann, Han, and Mitchell [497], who describe Algorithms 14.128 and 14.130. They also provide an analysis of the expected number of non-zero entries in an $SR(k)$ representation for a randomly selected t -bit exponent (see Note 14.132), as well as a complexity analysis of Algorithm 14.130 for various values of t and k . Lam and Hui [735] proposed an alternate string-replacement algorithm. The idea is to precompute all odd powers $g, g^3, g^5, \dots, g^{2^k-1}$ for some fixed positive integer k . Given a t -bit exponent e , start at the most significant bit, and look for the longest bitstring of bitlength at most k whose last digit is a 1 (i.e., this substring represents an odd positive integer between 1 and $2^k - 1$). Applying a left-to-right square-and-multiply exponentiation algorithm based on this scanning process results in an algorithm which requires, at most, $\lceil t/k \rceil$ multiplications. Lam and Hui proved that as t increases, the average number of multiplications approaches $\lceil t/(k+1) \rceil$.