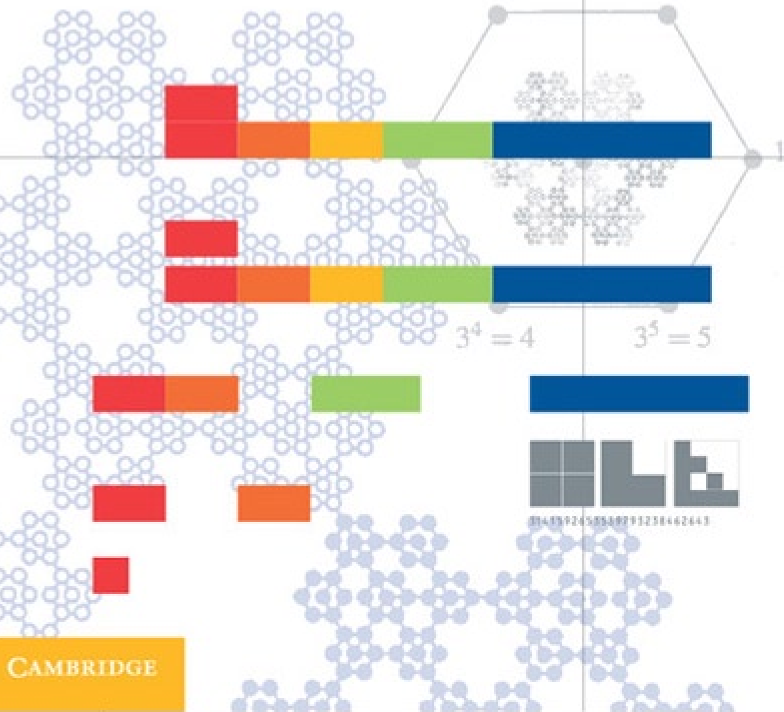
Modern Computer Algebra

Third Edition

Joachim von zur Gathen and Jürgen Gerhard



CAMBRIDGE

CAMBRIDGE

more information – www.cambridge.org/9781107039032

Modern Computer Algebra

Computer algebra systems are now ubiquitous in all areas of science and engineering. This highly successful textbook, widely regarded as the "bible of computer algebra", gives a thorough introduction to the algorithmic basis of the mathematical engine in computer algebra systems. Designed to accompany one- or two-semester courses for advanced undergraduate or graduate students in computer science or mathematics, its comprehensiveness and reliability has also made it an essential reference for professionals in the area.

Special features include: detailed study of algorithms including time analysis; implementation reports on several topics; complete proofs of the mathematical underpinnings; and a wide variety of applications (among others, in chemistry, coding theory, cryptography, computational logic, and the design of calendars and musical scales). A great deal of historical information and illustration enlivens the text.

In this third edition, errors have been corrected and much of the Fast Euclidean Algorithm chapter has been renovated.

Joachim von zur Gathen has a PhD from Universität Zürich and has taught at the University of Toronto and the University of Paderborn. He is currently a professor at the Bonn–Aachen International Center for Information Technology (B-IT) and the Department of Computer Science at Universität Bonn.

Jürgen Gerhard has a PhD from Universität Paderborn. He is now Director of Research at Maplesoft in Canada, where he leads research collaborations with partners in Canada, France, Russia, Germany, the USA, and the UK, as well as a number of consulting projects for global players in the automotive industry.

Modern Computer Algebra
Third Edition

JOACHIM VON ZUR GATHEN
Bonn–Aachen International Center
for Information Technology (B-IT)

JÜRGEN GERHARD
Maplesoft, Waterloo

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town,
Singapore, São Paulo, Delhi, Mexico City

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9781107039032

First and second editions © Cambridge University Press 1999, 2003
Third edition © Joachim von zur Gathen and Jürgen Gerhard 2013

This publication is in copyright. Subject to statutory exception
and to the provisions of relevant collective licensing agreements,
no reproduction of any part may take place without the written
permission of Cambridge University Press.

First published 1999
Second edition 2003
Third edition 2013.

Printed and bound by CPI Group (UK) Ltd, Croydon CR0 4YY

A catalogue record for this publication is available from the British Library

ISBN 978-1-107-03903-2 Hardback

Additional resources for this publication at http://cosec.bit.uni-bonn.de/science/mca

Cambridge University Press has no responsibility for the persistence or
accuracy of URLs for external or third-party internet websites referred to
in this publication, and does not guarantee that any content on such
websites is, or will remain, accurate or appropriate.

To Dorothea, Rafaela, Désirée
For endless patience

To Mercedes Cappuccino

Contents

Introduction **1**

1 Cyclohexane, cryptography, codes, and computer algebra **11**
1.1 Cyclohexane conformations 11
1.2 The RSA cryptosystem 16
1.3 Distributed data structures 18
1.4 Computer algebra systems 19

I Euclid **23**

2 Fundamental algorithms **29**
2.1 Representation and addition of numbers 29
2.2 Representation and addition of polynomials 32
2.3 Multiplication 34
2.4 Division with remainder 37
Notes 41
Exercises 41

3 The Euclidean Algorithm **45**
3.1 Euclidean domains 45
3.2 The Extended Euclidean Algorithm 47
3.3 Cost analysis for $\mathbb{Z}$ and $F[x]$ 51
3.4 (Non-)Uniqueness of the gcd 55
Notes 61
Exercises 62

4 Applications of the Euclidean Algorithm **69**
4.1 Modular arithmetic 69
4.2 Modular inverses via Euclid 73
4.3 Repeated squaring 75
4.4 Modular inverses via Fermat 76

4.5 Linear Diophantine equations . . . 77
4.6 Continued fractions and Diophantine approximation . . . 79
4.7 Calendars . . . 83
4.8 Musical scales . . . 84
Notes . . . 88
Exercises . . . 91

5 Modular algorithms and interpolation 97
5.1 Change of representation . . . 100
5.2 Evaluation and interpolation . . . 101
5.3 Application: Secret sharing . . . 103
5.4 The Chinese Remainder Algorithm . . . 104
5.5 Modular determinant computation . . . 109
5.6 Hermite interpolation . . . 113
5.7 Rational function reconstruction . . . 115
5.8 Cauchy interpolation . . . 118
5.9 Padé approximation . . . 121
5.10 Rational number reconstruction . . . 124
5.11 Partial fraction decomposition . . . 128
Notes . . . 131
Exercises . . . 132

6 The resultant and gcd computation 141
6.1 Coefficient growth in the Euclidean Algorithm . . . 141
6.2 Gauß' lemma . . . 147
6.3 The resultant . . . 152
6.4 Modular gcd algorithms . . . 158
6.5 Modular gcd algorithm in $F[x,y]$ . . . 161
6.6 Mignotte's factor bound and a modular gcd algorithm in $\mathbb{Z}[x]$ . . . 164
6.7 Small primes modular gcd algorithms . . . 168
6.8 Application: intersecting plane curves . . . 171
6.9 Nonzero preservation and the gcd of several polynomials . . . 176
6.10 Subresultants . . . 178
6.11 Modular Extended Euclidean Algorithms . . . 183
6.12 Pseudodivision and primitive Euclidean Algorithms . . . 190
6.13 Implementations . . . 193
Notes . . . 197
Exercises . . . 199

7 Application: Decoding BCH codes 209
Notes . . . 215
Exercises . . . 215

II Newton 217

8 Fast multiplication 221
8.1 Karatsuba's multiplication algorithm . . . 222
8.2 The Discrete Fourier Transform and the Fast Fourier Transform . . . 227
8.3 Schönhage and Strassen's multiplication algorithm . . . 238
8.4 Multiplication in $\mathbb{Z}[x]$ and $R[x,y]$ . . . 245
Notes . . . 247
Exercises . . . 248

9 Newton iteration 257
9.1 Division with remainder using Newton iteration . . . 257
9.2 Generalized Taylor expansion and radix conversion . . . 264
9.3 Formal derivatives and Taylor expansion . . . 265
9.4 Solving polynomial equations via Newton iteration . . . 267
9.5 Computing integer roots . . . 271
9.6 Newton iteration, Julia sets, and fractals . . . 273
9.7 Implementations of fast arithmetic . . . 278
Notes . . . 286
Exercises . . . 287

10 Fast polynomial evaluation and interpolation 295
10.1 Fast multipoint evaluation . . . 295
10.2 Fast interpolation . . . 299
10.3 Fast Chinese remaindering . . . 301
Notes . . . 306
Exercises . . . 306

11 Fast Euclidean Algorithm 313
11.1 A fast Euclidean Algorithm for polynomials . . . 313
11.2 Subresultants via Euclid's algorithm . . . 327
Notes . . . 332
Exercises . . . 332

12 Fast linear algebra 335
12.1 Strassen's matrix multiplication . . . 335
12.2 Application: fast modular composition of polynomials . . . 338
12.3 Linearly recurrent sequences . . . 340
12.4 Wiedemann's algorithm and black box linear algebra . . . 346
Notes . . . 352
Exercises . . . 353

13 Fourier Transform and image compression 359
13.1 The Continuous and the Discrete Fourier Transform 359
13.2 Audio and video compression 363
Notes 368
Exercises 368

III Gauß 371

14 Factoring polynomials over finite fields 377
14.1 Factorization of polynomials 377
14.2 Distinct-degree factorization 380
14.3 Equal-degree factorization: Cantor and Zassenhaus' algorithm . . 382
14.4 A complete factoring algorithm 389
14.5 Application: root finding 392
14.6 Squarefree factorization 393
14.7 The iterated Frobenius algorithm 398
14.8 Algorithms based on linear algebra 401
14.9 Testing irreducibility and constructing irreducible polynomials . 406
14.10 Cyclotomic polynomials and constructing BCH codes 412
Notes 417
Exercises 422

15 Hensel lifting and factoring polynomials 433
15.1 Factoring in $\mathbb{Z}[x]$ and $\mathbb{Q}[x]$: the basic idea 433
15.2 A factoring algorithm 435
15.3 Frobenius' and Chebotarev's density theorems 441
15.4 Hensel lifting 444
15.5 Multifactor Hensel lifting 450
15.6 Factoring using Hensel lifting: Zassenhaus' algorithm 453
15.7 Implementations 461
Notes 465
Exercises 467

16 Short vectors in lattices 473
16.1 Lattices 473
16.2 Lenstra, Lenstra and Lovász' basis reduction algorithm 475
16.3 Cost estimate for basis reduction 480
16.4 From short vectors to factors 487
16.5 A polynomial-time factoring algorithm for $\mathbb{Z}[x]$ 489
16.6 Factoring multivariate polynomials 493
Notes 496
Exercises 498

17 Applications of basis reduction **503**
17.1 Breaking knapsack-type cryptosystems 503
17.2 Pseudorandom numbers 505
17.3 Simultaneous Diophantine approximation 505
17.4 Disproof of Mertens' conjecture 508
Notes 509
Exercises 509

IV Fermat **511**

18 Primality testing **517**
18.1 Multiplicative order of integers 517
18.2 The Fermat test 519
18.3 The strong pseudoprimality test 520
18.4 Finding primes 523
18.5 The Solovay and Strassen test 529
18.6 Primality tests for special numbers 530
Notes 531
Exercises 534

19 Factoring integers **541**
19.1 Factorization challenges 541
19.2 Trial division 543
19.3 Pollard's and Strassen's method 544
19.4 Pollard's rho method 545
19.5 Dixon's random squares method 549
19.6 Pollard's $p-1$ method 557
19.7 Lenstra's elliptic curve method 557
Notes 567
Exercises 569

20 Application: Public key cryptography **573**
20.1 Cryptosystems 573
20.2 The RSA cryptosystem 576
20.3 The Diffie–Hellman key exchange protocol 578
20.4 The ElGamal cryptosystem 579
20.5 Rabin's cryptosystem 579
20.6 Elliptic curve systems 580
Notes 580
Exercises 580

V Hilbert 585

21 Gröbner bases 591
21.1 Polynomial ideals . . . 591
21.2 Monomial orders and multivariate division with remainder . . . 595
21.3 Monomial ideals and Hilbert's basis theorem . . . 601
21.4 Gröbner bases and S-polynomials . . . 604
21.5 Buchberger's algorithm . . . 608
21.6 Geometric applications . . . 612
21.7 The complexity of computing Gröbner bases . . . 616
Notes . . . 617
Exercises . . . 619

22 Symbolic integration 623
22.1 Differential algebra . . . 623
22.2 Hermite's method . . . 625
22.3 The method of Lazard, Rioboo, Rothstein, and Trager . . . 627
22.4 Hyperexponential integration: Almkvist & Zeilberger's algorithm 632
Notes . . . 640
Exercises . . . 641

23 Symbolic summation 645
23.1 Polynomial summation . . . 645
23.2 Harmonic numbers . . . 650
23.3 Greatest factorial factorization . . . 653
23.4 Hypergeometric summation: Gosper's algorithm . . . 658
Notes . . . 669
Exercises . . . 671

24 Applications 677
24.1 Gröbner proof systems . . . 677
24.2 Petri nets . . . 679
24.3 Proving identities and analysis of algorithms . . . 681
24.4 Cyclohexane revisited . . . 685
Notes . . . 697
Exercises . . . 698

Appendix 701

25 Fundamental concepts 703
25.1 Groups . . . 703
25.2 Rings . . . 705

25.3 Polynomials and fields 708
25.4 Finite fields 711
25.5 Linear algebra 713
25.6 Finite probability spaces 717
25.7 “Big Oh” notation 720
25.8 Complexity theory 721
Notes 724

Sources of illustrations 725
Sources of quotations 725
List of algorithms 730
List of figures and tables 732
References 734
List of notation 768
Index 769

Keeping up to date

Addenda and corrigenda, comments, solutions to selected exercises, and ordering information can be found on the book’s web page:

`http://cosec.bit.uni-bonn.de/science/mca/`

A Beggar's Book Out-worths a Noble's Blood.[1]

William Shakespeare (1613)

Some books are to be tasted, others to be swallowed,
and some few to be chewed and digested.

Francis Bacon (1597)

Les plus grands analystes eux-mêmes ont bien rarement dédaigné de se tenir à la portée de la classe *moyenne* des lecteurs; elle est en effet la plus nombreuse, et celle qui a le plus à profiter dans leurs écrits.[2]

Anonymous referee (1825)

It is true, we have already a great many Books of *Algebra*,
and one might even furnish a moderate Library
purely with Authors on that Subject.

Isaac Newton (1728)

فحررت هذا الكتاب وجمعت فيه جميع ما يحتاج اليه الحاسب
محترزا عن اشباع ممل و اختصار مخل[3]

Ghiyāth al-Dīn Jamshīd bin Masʿūd bin Maḥmūd al-Kāshī (1427)

[1] The sources for the quotations are given on pages 725–729.

[2] The greatest analysts [mathematicians] themselves have rarely shied away from keeping within the reach of the *average* class of readers; this is in fact the most numerous one, and the one that stands to profit most from their writing.

[3] I wrote this book and compiled in it everything that is necessary for the computer, avoiding both boring verbosity and misleading brevity.

Introduction

In science and engineering, a successful attack on a problem will usually lead to some equations that have to be solved. There are many types of such equations: differential equations, linear or polynomial equations or inequalities, recurrences, equations in groups, tensor equations, etc. In principle, there are two ways of solving such equations: approximately or exactly. *Numerical analysis* is a well-developed field that provides highly successful mathematical methods and computer software to compute *approximate* solutions.

Computer algebra is a more recent area of computer science, where mathematical tools and computer software are developed for the *exact* solution of equations.

Why use approximate solutions at all if we can have exact solutions? The answer is that in many cases an exact solution is not possible. This may have various reasons: for certain (simple) ordinary differential equations, one can prove that no closed form solution (of a specified type) is possible. More important are questions of efficiency: any system of linear equations, say with rational coefficients, can be solved exactly, but for the huge linear systems that arise in meteorology, nuclear physics, geology or other areas of science, only approximate solutions can be computed efficiently. The exact methods, run on a supercomputer, would not yield answers within a few days or weeks (which is not really acceptable for weather prediction).

However, within its range of exact solvability, computer algebra usually provides more interesting answers than traditional numerical methods. Given a differential equation or a system of linear equations with a parameter t, the scientist gets much more information out of a closed form solution in terms of t than from several solutions for specific values of t.

Many of today's students may not know that the *slide rule* was an indispensable tool of engineers and scientists until the 1960s. *Electronic pocket calculators* made them obsolete within a short time. In the coming years, *computer algebra systems* will similarly replace calculators for many purposes. Although still bulky and expensive (hand-held computer algebra calculators are yet a novelty), these systems can easily perform exact (or arbitrary precision) arithmetic with numbers,

matrices, polynomials, etc. They will become an indispensable tool for the scientist and engineer, from students to the work place. These systems are now becoming integrated with other software, like numerical packages, CAD/CAM, and graphics.

The goal of this text is to give an introduction to the basic methods and techniques of computer algebra. Our focus is threefold:

- complete presentation of the mathematical underpinnings,
- asymptotic analysis of our algorithms, sometimes "Oh-free",
- development of asymptotically fast methods.

It is customary to give bounds on running times of algorithms (if any are given at all) in a "big-Oh" form (explained in Section 25.7), say as $O(n \log n)$ for the FFT. We often prove "Oh-free" bounds in the sense that we identify the numerical coefficient of the leading term, as $\frac{3}{2} n \log_2 n$ in the example; we may then add $O(\text{smaller terms})$. But we have not played out the game of minimizing these coefficients; the reader is encouraged to find smaller constants herself.

Many of these fast methods have been known for a quarter of a century, but their impact on computer algebra systems has been slight, partly due to an "unfortunate myth" (Bailey, Lee & Simon 1990) about their practical (ir)relevance. But their usefulness has been forcefully demonstrated in the last few years; we can now solve problems—for example, the factorization of polynomials—of a size that was unassailable a few years ago. We expect this success to expand into other areas of computer algebra, and indeed hope that this text may contribute to this development. The full treatment of these fast methods motivates the "modern" in its title. (Our title is a bit risqué, since even a "modern" text in a rapidly evolving discipline such as ours will obsolesce quickly.)

The basic objects of computer algebra are numbers and polynomials. Throughout the text, we stress the structural and algorithmic similarities between these two domains, and also where the similarities break down. We concentrate on polynomials, in particular univariate polynomials over a field, and pay special attention to finite fields.

We will consider arithmetic algorithms in some basic domains. The tasks that we will analyze include conversion between representations, addition, subtraction, multiplication, division, division with remainder, greatest common divisors, and factorization. The domains of fundamental importance for computer algebra are the natural numbers, the rational numbers, finite fields, and polynomial rings.

Our three goals, as stated above, are too ambitious to keep up throughout. In some chapters, we have to content ourselves with sketches of methods and outlooks on further results. Due to space limitations, we sometimes have recourse to the lamentable device of "leaving the proof to the reader". Don't worry, be happy: solutions to the corresponding exercises are available on the book's web site.

After writing most of the material, we found that we could structure the book into five parts, each named after a mathematician that made a pioneering contribution on which some (but, of course, not all) of the modern methods in the respective part rely. In each part, we also present selected applications of some of the algorithmic methods.

The first part EUCLID examines Euclid's algorithm for calculating the gcd, and presents the subresultant theory for polynomials. Applications are numerous: modular algorithms, continued fractions, Diophantine approximation, the Chinese Remainder Algorithm, secret sharing, and the decoding of BCH codes.

The second part NEWTON presents the basics of fast arithmetic: FFT-based multiplication, division with remainder and polynomial equation solving via Newton iteration, and fast methods for the Euclidean Algorithm and the solution of systems of linear equations. The FFT originated in signal processing, and we discuss one of its applications, image compression.

The third part GAUSS deals exclusively with polynomial problems. We start with univariate factorization over finite fields, and include the modern methods that make attacks on enormously large problems feasible. Then we discuss polynomials with rational coefficients. The two basic algorithmic ingredients are Hensel lifting and short vectors in lattices. The latter has found many applications, from breaking certain cryptosystems to Diophantine approximation.

The fourth part FERMAT is devoted to two integer problems that lie at the foundation of algorithmic number theory: primality testing and factorization. The most famous modern application of these classical topics is in public key cryptography.

The fifth part HILBERT treats three different topics which are somewhat more advanced than the rest of the text, and where we can only exhibit the foundations of a rich theory. The first area is Gröbner bases, a successful approach to deal with multivariate polynomials, in particular questions about common roots of several polynomials. The next topic is symbolic integration of rational and hyperexponential functions. The final subject is symbolic summation; we discuss polynomial and hypergeometric summation.

The text concludes with an appendix that presents some foundational material in the language we use throughout the book: The basics of groups, rings, and fields, linear algebra, probability theory, asymptotic O-notation, and complexity theory.

Each of the first three parts contains an implementation report on some of the algorithms presented in the text. As case studies, we use two special purpose packages for integer and polynomial arithmetic: NTL by Victor Shoup and BIPOLAR by the authors.

Most chapters end with some bibliographical and historical notes or supplementary remarks, and a variety of exercises. The latter are marked according to their difficulty: exercises with a * are somewhat more advanced, and the few marked with ** are more difficult or may require material not covered in the text.

Laborious (but not necessarily difficult) exercises are marked by a long arrow $\longrightarrow$. The book's web page `http://cosec.bit.uni-bonn.de/science/mca/` provides some solutions.

This book presents foundations for the mathematical engine underlying any computer algebra system, and we give substantial coverage—often, but not always, up to the state of the art—for the material of the first three parts, dealing with Euclid's algorithm, fast arithmetic, and the factorization of polynomials. But we hasten to point out some unavoidable shortcomings. For one, we cannot cover completely even those areas that we discuss, and our treatment leaves out major interesting developments in the areas of computational linear algebra, sparse multivariate polynomials, combinatorics and computational number theory, quantifier elimination and solving polynomial equations, and differential and difference equations. Secondly, some important questions are left untouched at all; we only mention computational group theory,parallel computation, computing with transcendental functions, isolating real and complex roots of polynomials, and the combination of symbolic and numeric methods. Finally, a successful computer algebra system involves much more than just the mathematical engine: efficient data structures, a fast kernel and a large compiled or interpreted library, user interface, graphics capability, interoperability of software packages, clever marketing, etc. These issues are highly technology-dependent, and there is no single good solution for them.

The present book can be used as the textbook for a one-semester or a two-semester course in computer algebra. The basic arithmetic algorithms are discussed in Chapters 2 and 3, and Sections 4.1–4.4, 5.1–5.5, 8.1–8.2, 9.1–9.4, 14.1–14.6, and 15.1–15.2. In addition, a one-semester undergraduate course might be slanted towards computational number theory (9.5, 18.1–18.4, and parts of Chapter 20), geometry (21.1–21.6), or integration (4.5, 5.11, 6.2–6.4, and Chapter 22), supplemented by fun applications from 4.6–4.8, 5.6–5.9, 6.8, 9.6, Chapter 13, and Chapters 1 and 24. A two-semester course could teach the "basics" and 6.1–6.7, 10.1–10.2, 15.4–15.6, 16.1–16.5, 18.1–18.3, 19.1–19.2, 19.4, 19.5 or 19.6–19.7, and one or two of Chapters 21–23, maybe with some applications from Chapters 17, 20, and 24. A graduate course can be more eclectic. We once taught a course on "factorization", using parts of Chapters 14–16 and 19. Another possibility is a graduate course on "fast algorithms" based on Part II. For any of these suggestions, there is enough material so that an instructor will still have plenty of choice of which areas to skip. The logical dependencies between the chapters are given in Figure 1.

The prerequisite for such a course is linear algebra and a certain level of mathematical maturity; particularly useful is a basic familiarity with algebra and analysis of algorithms. However, to allow for the large variations in students' background, we have included an appendix that presents the necessary tools. For that material, the borderline between the boring and the overly demanding varies too much

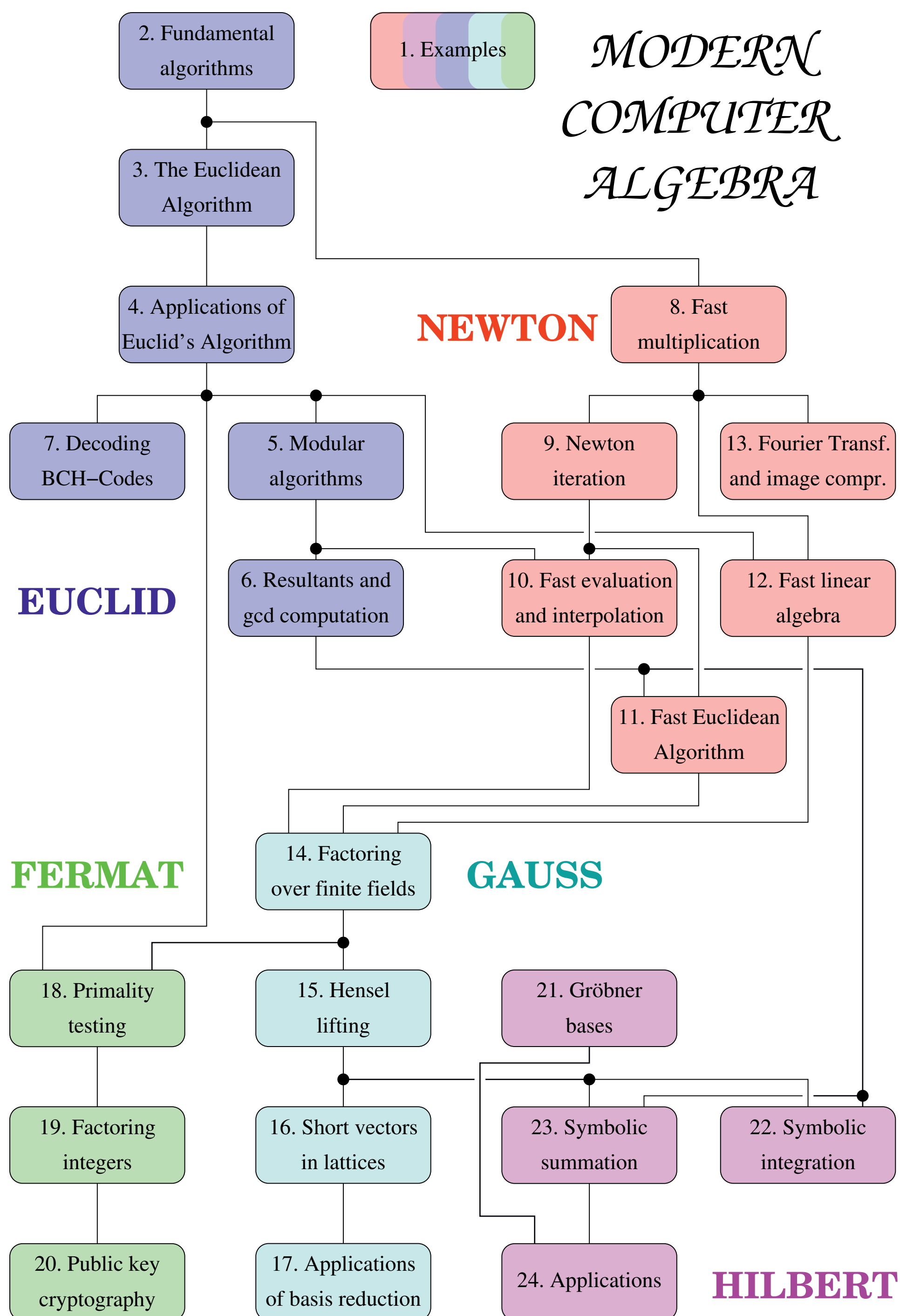


FIGURE 1: Leitfaden.

to get it right for everyone. If those notions and tools are unfamiliar, an instructor may have to expand beyond the condensed description in the appendix. Otherwise, most of the presentation is self-contained, and the exceptions are clearly indicated. By their nature, some of the applications assume a background in the relevant area.

The beginning of each part presents a biographical sketch of the scientist after which it is named, and throughout the text we indicate some of the origins of our material. For lack of space and competence, this is not done in a systematic way, let alone with the goal of completeness, but we do point to some early sources, often centuries old, and quote some of the original work. Interest in such historical issues is, of course, a matter of taste. It is satisfying to see how many algorithms are based on venerable methods; our essentially "modern" aspect is the concern with asymptotic complexity and running times, faster and faster algorithms, and their computer implementation.

Acknowledgements. This material has grown from undergraduate and graduate courses that the first author has taught over more than a decade in Toronto, Zürich, Santiago de Chile, Canberra, and Paderborn. He wants to thank above all his two teachers: Volker Strassen, who taught him mathematics, and Allan Borodin, who taught him computer science. To his friend Erich Kaltofen he is grateful for many enlightening discussions about computer algebra.

The second author wants to thank his two supervisors, Helmut Meyn and Volker Strehl, for many stimulating lectures in computer algebra.

The support and enthusiasm of two groups of people have made the courses a pleasure to teach. On the one hand, the colleagues, several of whom actually shared in the teaching: Leopoldo Bertossi, Allan Borodin, Steve Cook, Faith Fich, Shuhong Gao, John Lipson, Mike Luby, Charlie Rackoff, and Victor Shoup. On the other hand, lively groups of students took the courses, solved the exercises and tutored others about them, and some of them were the scribes for the course notes that formed the nucleus of this text. We thank particularly Paul Beame, Isabel de Correa, Wayne Eberly, Mark Giesbrecht, Rod Glover, Silke Hartlieb, Jim Hoover, Keju Ma, Jim McInnes, Pierre McKenzie, Sun Meng, Rob Morenz, Michael Nöcker, Daniel Panario, Michel Pilote, and François Pitt.

Thanks for help on various matters go to Eric Bach, Peter Blau, Wieb Bosma, Louis Bucciarelli, Désirée von zur Gathen, Keith Geddes, Dima Grigoryev, Johan Håstad, Dieter Herzog, Marek Karpinski, Wilfrid Keller, Les Klinger, Werner Krandick, Ton Levelt, János Makowsky, Ernst Mayr, François Morain, Gerry Myerson, Michael Nüsken, David Pengelley, Bill Pickering, Tomás Recio, Jeff Shallit, Igor Shparlinski, Irina Shparlinski, and Paul Zimmermann.

We thank Sandra Feisel, Carsten Keller, Thomas Lücking, Dirk Müller, and Olaf Müller for programming and the substantial task of producing the index, and Marianne Wehry for tireless help with the typing.

We are indebted to Sandra Feisel, Adalbert Kerber, Preda Mihăilescu, Michael Nöcker, Daniel Panario, Peter Paule, Daniel Reischert, Victor Shoup, and Volker Strehl for carefully proofreading parts of the draft.

Paderborn, January 1999

The 2003 edition. The great French mathematician Pierre Fermat never published a thing in his lifetime. One of the reasons was that in his days, books and other publications often suffered vitriolic attacks for perceived errors, major or minor, frequently combined with personal slander.

Our readers are friendlier. They pointed out about 160 errors and possible improvements in the 1999 edition to us, but usually sugared their messages with sweet compliments. Thanks, friends, for helping us feel good and produce a better book now! We gratefully acknowledge the assistance of Sergeĭ Abramov, Michael Barnett, Andreas Beschorner, Murray Bremner, Peter Bürgisser, Michael Clausen, Rob Corless, Abhijit Das, Ruchira Datta, Wolfram Decker, Emrullah Durucan, Friedrich Eisenbrand, Ioannis Emiris, Torsten Fahle, Benno Fuchssteiner, Rod Glover, David Goldberg, Mitch Harris, Dieter Herzog, Andreas Hirn, Mark van Hoeij, Dirk Jung, Kyriakos Kalorkoti, Erich Kaltofen, Karl-Heinz Kiyek, Andrew Klapper, Don Knuth, Ilias Kotsireas, Werner Krandick, Daniel Lauer, Daniel Bruce Lloyd, Martin Lotz, Thomas Lücking, Heinz Lüneburg, Mantsika Matooane, Helmut Meyn, Eva Mierendorff, Daniel Müller, Olaf Müller, Seyed Hesameddin Najafi, Michael Nöcker, Michael Nüsken, Andreas Oesterhelt, Daniel Panario, Thilo Pruschke, Arnold Schönhage, Jeff Shallit, Hans Stetter, David Theiwes, Thomas Viehmann, Volker Weispfenning, Eugene Zima, and Paul Zimmermann.

Our thanks also go to Christopher Creutzig, Katja Daubert, Torsten Metzner, Eva Müller, Peter Serocka, and Marianne Wehry.

Besides correcting the known errors and (unintentionally) introducing new ones, we smoothed and updated various items, and made major changes in Chapters 3, 15, and 22.

Paderborn, February 2002

The 2013 edition. Many people have implemented algorithms from this text and were happy with it. A few have tried their hands at the fast Euclidean algorithm from Chapter 11 and became unhappy. No wonder — the description contained a bug which squeezed through an unseen crack in our proof of correctness. That particular crack has been sealed for the present edition, and in fact much of Chapter 11 is renovated. In addition, about 80 other errors have been corrected. Thanks go to John R. Black, Murray Bremner, Winfried Bruns, Evan Jingchi Chen, Howard Cheng, Stefan Dreker, Olav Geil, Giulio Genovese, Stefan Gerhold, Charles-Antoine Giuliani, Sebastian Grimsell, Masaaki Kanno, Tom Koornwinder, Heiko Körner, Volker Krummel, Martina Kuhnert, Jens Kunerle,

Eugene Luks, Olga Mendoza, Helmut Meyn, Guillermo Moreno-Socías, Olaf Müller, Peter Nilsson, Michael Nüsken, Kathy Pinzon, Robert Schwarz, Jeff Shallit, Viktor Shoup, Allan Steel, Fre Vercauteren, Paul Vrbik, Christiaan van de Woestijne, Huang Yong, Konstantin Ziegler, and Paul Zimmermann for their hints. We also acknowledge the help of Martina Kuhnert and Michael Nüsken in producing this edition.

Separate errata pages for each edition will be kept on the book's website `http://cosec.bit.uni-bonn.de/science/mca/`.

Dear readers, the hunt for errors is not over. Please keep on sending them to us at `gathen@bit.uni-bonn.de` or `gerhard.juergen@web.de`. And while hunting, enjoy the reading!

Bonn and Georgetown, January 2013

Note. We produced the postscript files for this book with the invaluable help of the following software packages: Leslie Lamport's LaTeX, based on Don Knuth's TeX, Klaus Lagally's ArabTeX, Oren Patashnik's BibTeX, Pehong Chen's MakeIndex, MAPLE, MUPAD, Victor Shoup's NTL, Thomas Williams' and Colin Kelley's gnuplot, the Persistence of Vision Ray Tracer POV-Ray, and xfig.

Clarke's Third Law:
Any sufficiently advanced technology is indistinguishable from magic.
Arthur C. Clarke (c. 1969)

L'avancement et la perfection des mathématiques
sont intimement liés à la prospérité de l'État.[1]
Napoléon I. (1812)

It must be easy [. . .] to bring out a *double* set of *results*, viz. —1st,
the *numerical magnitudes* which are the results of operations
performed on *numerical data.* [. . .] 2ndly, the *symbolical results*
to be attached to those numerical results, which symbolical results
are not less the necessary and logical consequences
of operations performed upon *symbolical data*,
than are numerical results when the data are numerical.
Augusta Ada Lovelace (1843)

There are too goddamned many machines that spew out data too fast.
Robert Ludlum (1995)

After all, the whole purpose of science is not technology—
God knows we have gadgets enough already.
Eric Temple Bell (1937)

[1] The advancement and perfection of mathematics are intimately connected with the prosperity of the State.

1

Cyclohexane, cryptography, codes, and computer algebra

Three examples in this chapter illustrate some applications of the ideas and methods of computer algebra: the spatial configurations (conformations) of the cyclohexane molecule, a chemical problem with an intriguing geometric solution; a cryptographic protocol for the secure transmission of messages; and distributed codes for sharing secrets or sending packets over a faulty network. Throughout this book you will find such sample applications in a wide variety of areas, from the design of calendars and musical scales to image compression and the intersection of algebraic curves. The last section in this chapter gives a concise overview of some computer algebra systems.

1.1. Cyclohexane conformations

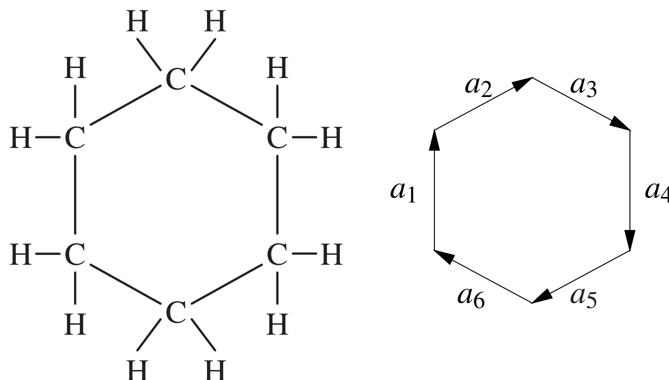


FIGURE 1.1: The structure formula for cyclohexane (C_6H_{12}), and the orientation we give to the bonds $a_1, \ldots, a_6$.

We start with an example from chemistry. It illustrates the three typical steps in mathematical applications: creating a mathematical model of the problem at hand, "solving" the model, and interpreting the solution in the original problem. Usually,

none of these steps is straightforward, and one often has to go back and modify the approach.

Cyclohexane C_6H_{12} (Figure 1.1), a molecule from organic chemistry, is a hydrocarbon consisting of six carbon atoms (C) connected to each other in a cycle and twelve hydrogen atoms (H), two attached to each carbon atom. The four bonds of one carbon atom (two bonds to adjacent carbon atoms and two bonds to hydrogen atoms) are arranged in the form of a tetrahedron, with the carbon in the center and its bonds pointing to the four corners. The angle α between any two bonds is about 109 degrees (the precise value of α satisfies $\cos\alpha = -1/3$). Two adjacent carbon atoms may freely rotate around the bond between them.

Chemists have observed that cyclohexane occurs in two incongruent conformations (which are not transformable into each other by rotations and reflections), a “chair” (Figure 1.2) and a “boat” (Figure 1.3), and experiments have shown that the “chair” occurs far more frequently than the “boat”. The frequency of occurrence of a conformation depends on its free energy—a general rule is that molecules try to minimize the free energy—which in turn depends on the spatial structure.

When modeling the molecule by means of plastic tubes (Figure 1.4) representing the carbon atoms and the bonds between them (omitting the hydrogen atoms for simplicity) in such a way that rotations around the bonds are possible, one observes that there is a certain amount of freedom in moving the atoms by rotations around the bonds in the “boat” conformation (we will call it the **flexible** conformation), but that the “chair” conformation is **rigid**, and that it appears to be impossible to get from the “boat” to the “chair” conformation. Can we mathematically model and, if possible, explicitly describe this behavior?

We let $a_1, \ldots, a_6 \in \mathbb{R}^3$ be the orientations of the six bonds in three-space, so that all six vectors point in the same direction around the cyclic structure (Figure 1.1), and normalize the distance between two adjacent carbon atoms to be one. By $u \star v = u_1v_1 + u_2v_2 + u_3v_3$ we denote the usual inner product of two vectors $u = (u_1, u_2, u_3)$ and $v = (v_1, v_2, v_3)$ in $\mathbb{R}^3$. The cosine theorem says that $u \star v = \|u\|_2 \cdot \|v\|_2 \cdot \cos\beta$, where $\|u\|_2 = (u \star u)^{1/2}$ is the Euclidean norm and $\beta \in [0, \pi]$ is the angle between u and v, when both vectors are rooted at the origin. The above conditions then lead to the following system of equations:

$$
\begin{aligned}
a_1 \star a_1 = a_2 \star a_2 = \cdots = a_6 \star a_6 &= 1, \\
a_1 \star a_2 = a_2 \star a_3 = \cdots = a_6 \star a_1 &= \frac{1}{3}, \\
a_1 + a_2 + \cdots + a_6 &= 0.
\end{aligned}
\tag{1}
$$

The first line says that the length of each bond is 1. The second line expresses the fact that the angle between two bonds adjacent to the same carbon atom is α (the cosine is $1/3$ instead of $-1/3$ since, seen from the carbon atom, the two bonds

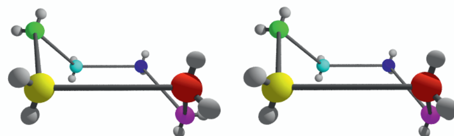


FIGURE 1.2: A stereo image of a “chair” conformation of cyclohexane. To see a three-dimensional image, hold the two pictures right in front of your eyes. Then relax your eyes and do not focus at the foreground, so that the two pictures fade away and each of them splits into two separate images (one for each eye). By further relaxing your eyes, try to match the right image that your left eye sees with the left image that your right eye sees. Now you see three images, each of the two outer ones only with one of your eyes, and the middle one with both eyes. Cautiously focus on the middle image, at the same time slowly moving the book away from your head, until the image is sharp. (This works best without wearing glasses.)

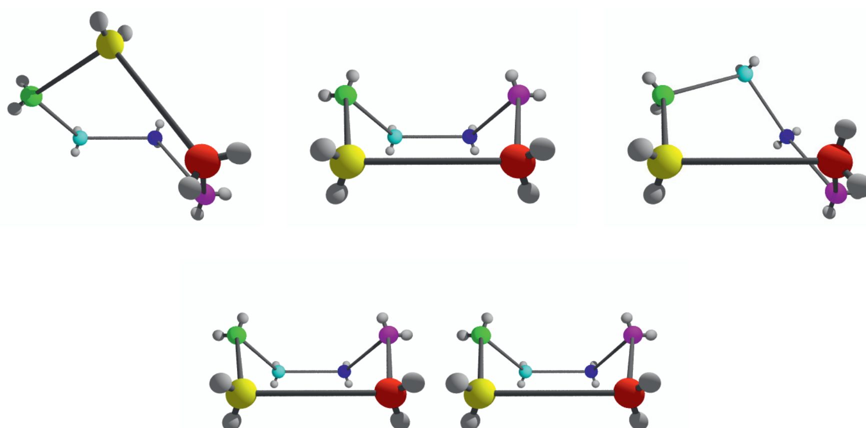


FIGURE 1.3: Three “boat” conformations of cyclohexane and a stereo image of the middle one (see Figure 1.2 for a viewing instruction).

have opposite orientation). Finally, the last line expresses the cyclic nature of the structure.

Together, (1) comprises $6+6+1=13$ equations in the 18 coordinates of the points $a_1,\dots,a_6$. The first ones are quadratic, and the last ones are linear. There is still redundancy coming from the whole structure’s possibility to move and rotate around freely in three-space. One possibility to remedy this is to introduce three more equations expressing the fact that a_1 and a_2 are parallel to the x-axis respectively the x,y-plane. These equations can be solved with a computer algebra system, but the resulting description of the solutions is highly complicated and non-intuitive.

FIGURE 1.4: A plumbing knee model of cyclohexane, with nearly right angles.

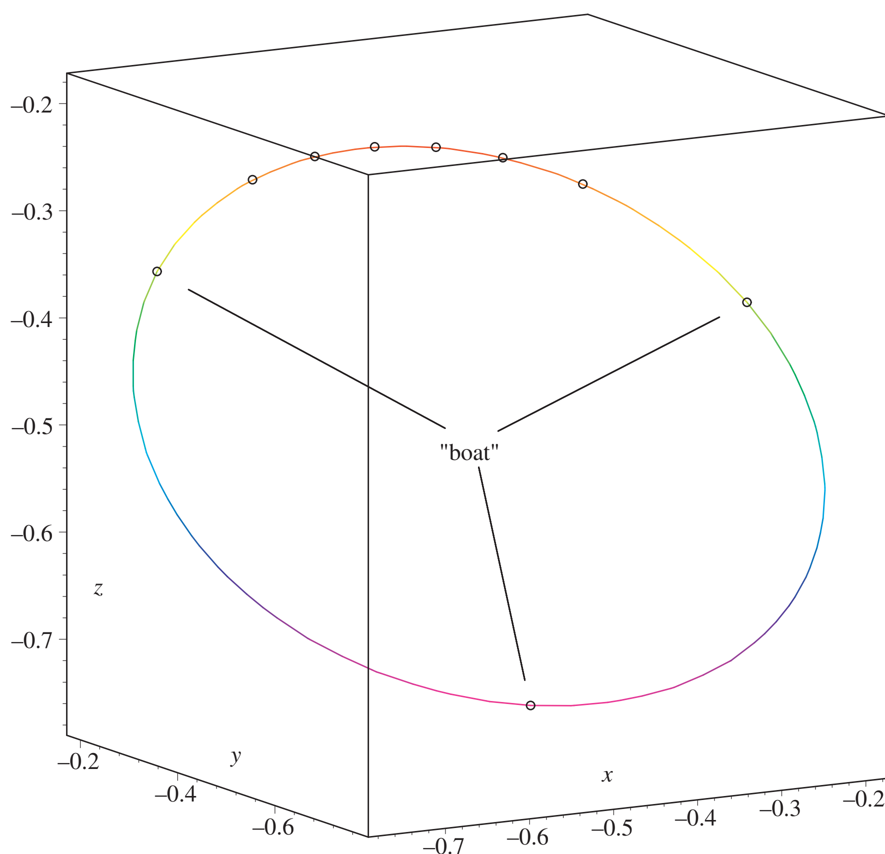


FIGURE 1.5: The curve E.

For a successful solution, we pursue a different, more symmetric approach, by taking the inner products $S_{ij} = a_i \star a_j$ for $1 \leq i, j \leq 6$ as unknowns instead of the coordinates of $a_1, \ldots, a_6$. This is described in detail in Section 24.4. Under the conditions (1), S_{ij} is the cosine of the angle between a_i and a_j. It turns out that all S_{ij} depend linearly on S_{13}, S_{35}, and S_{51}, and that the triples of values $(S_{13}, S_{35}, S_{51}) \in \mathbb{R}^3$ leading to the flexible conformations are given by the space curve E in Figure 1.5. The solution makes heavy use of various computer algebra tools, such as **resultants** (Chapter 6), **polynomial factorization** (Part III), and **Gröbner bases** (Chapter 21). The three marked points $(-1/3, -1/3, -7/9)$, $(-1/3, -7/9, -1/3)$ and $(-7/9, -1/3, -1/3)$ correspond to the three "boat" conformations in Figure 1.3 (all of them are equivalent by cyclically permuting $a_1, \ldots, a_6$). Actually, some information gets lost in the transition from the a_i to the S_{ij}, and each point on the curve E corresponds to precisely two spatial conformations which are mirror images of each other. The rigid conformation corresponds to the isolated solution $S_{13} = S_{35} = S_{51} = -1/3$ not lying on the curve.

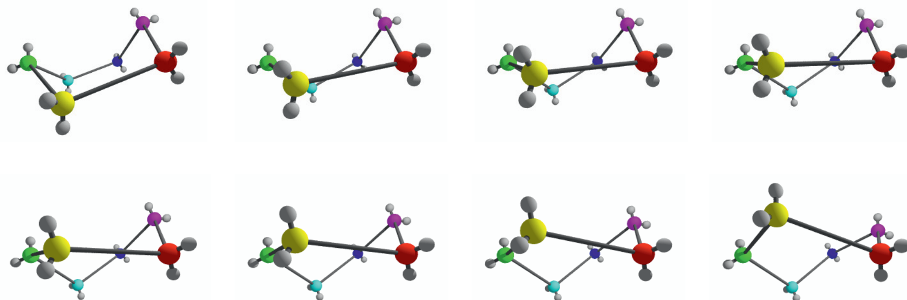
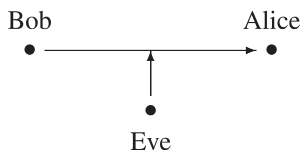


FIGURE 1.6: Eight flexible conformations of cyclohexane corresponding to the eight points marked in Figure 1.5. The first and the eighth ones are "boats". The point of view is such that the positions of the red, green, and blue carbon atoms are invariant for all eight pictures.

We built the simple physical "model" in Figure 1.4 of something similar to cyclohexane as follows. We bought six plastic plumbing "knees", with approximately a right angle. (German plumbing knees actually have an angle of about 93 degrees, for some deep hydrodynamic reason.) This differs considerably from the 109 degrees of the carbon tetrahedron, but on the other hand, it only cost about € 7. We stuck the six parts together and pulled an elastic cord through them to keep them from falling apart. Then one can smoothly turn the structure through the flexible conformations corresponding to the curve in Figure 1.5, physically "feeling" the curve. Pulling the whole thing forcibly apart, one can also get into the "chair" position. Now no wiggling or gentle twisting will move the structure; it is quite rigid.

1.2. The RSA cryptosystem

The basic scenario for cryptography is as follows. Bob wants to send a message to Alice in such a way that an eavesdropper (Eve) listening to the transmission channel cannot understand the message. This is done by enciphering the message so that only Alice, possessing the right **key**, can decipher it, but Eve, having no access to the key, has no chance to recover the message.



In classical **symmetric** cryptosystems, Alice and Bob use the same key for both encryption and decryption. The **RSA cryptosystem**, described in detail in

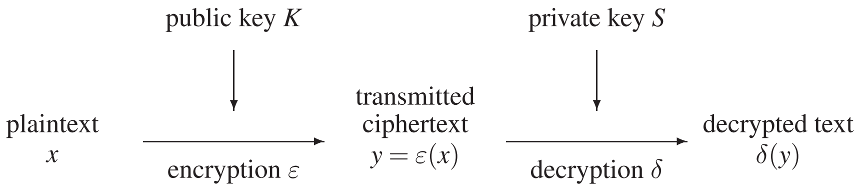


FIGURE 1.7: A public key cryptosystem.

Section 20.2, is an **asymmetric** or **public key cryptosystem**. Alice has a **public key** K that she publishes in some directory, and a **private key** S that she keeps secret. To encrypt a message, Bob uses Alice's public key, and only Alice can decrypt the ciphertext using her private key (Figure 1.7).

The RSA system works as follows. First, Alice randomly chooses two large (150 digit, say) prime numbers $p \neq q$ and computes their product $N = pq$. Efficient probabilistic **primality tests** (Chapter 18) make it easy to find such primes and then N, but the problem of finding p and q, when just N is given (that is, factoring N), seems very hard (see Chapter 19). Then she randomly chooses another integer $e \in \{2, \ldots, \varphi(N) - 2\}$ coprime to $\varphi(N)$, where φ is **Euler's totient function** (Section 4.2). For our particular N, the **Chinese Remainder Theorem** 5.3 implies that $\varphi(N) = (p-1) \cdot (q-1)$. Then Alice publishes the pair $K = (N, e)$. To obtain her private key, Alice uses the **Extended Euclidean Algorithm** 3.14 to compute $d \in \{2, \ldots, \varphi(N) - 2\}$ such that $ed \equiv 1 \bmod \varphi(N)$, and $S = (N, d)$ is her private key. Thus $(ed - 1)/\varphi(N)$ is an integer.

Before they can exchange messages, both parties agree upon a way of encoding messages (pieces of text) as integers in the range $0, \ldots, N-1$ (this is not part of the cryptosystem itself). For example, if messages are built from the 26 letters A through Z, we might identify A with 0, B with 1, …, Z with 25, and use the 26-ary representation for encoding. The message "CAESAR" is then encoded as

$$2 \cdot 26^0 + 0 \cdot 26^1 + 4 \cdot 26^2 + 18 \cdot 26^3 + 0 \cdot 26^4 + 17 \cdot 26^5 = 202\,302\,466.$$

Long messages are broken into pieces. Now Bob wants to send a message $x \in \{0, \ldots, N-1\}$ to Alice that only she can read. He looks up her public key (N, e), computes the encryption $y = \varepsilon(x) \in \{0, \ldots, N-1\}$ of x such that $y \equiv x^e \bmod N$, and sends y. Computing y can be done very efficiently using **repeated squaring** (Algorithm 4.8). To decrypt y, Alice uses her private key (N, d) to compute the decryption $x^* = \delta(y) \in \{0, \ldots, N-1\}$ of y with $x^* \equiv y^d \bmod N$. Now **Euler's theorem** (Section 18.1) says that $x^{\varphi(N)} \equiv 1 \bmod N$, if x and N are coprime. Thus

$$x^* \equiv y^d \equiv x^{ed} = x \cdot (x^{\varphi(N)})^{(ed-1)/\varphi(N)} \equiv x \bmod N,$$

and it follows that $x^* = x$ since x and x^* are both in $\{0,\ldots,N-1\}$. In fact, $x^* = x$ also holds when x and N have a nontrivial common divisor.

Without knowledge of d, however, it seems currently infeasible to compute x from N, e, and y. The only known way to do this is to factor N into its prime factors, and then to compute d with the Extended Euclidean Algorithm as Alice did, but **factoring integers** (Chapter 19) is extremely time-consuming: 300 digit numbers are beyond the capabilities of currently known factoring algorithms even on modern supercomputers or workstation networks.

Software packages like PGP ("Pretty Good Privacy"; see Zimmermann (1996) and http://www.openpgp.org) use the RSA cryptosystem for encrypting and authenticating e-mail and data files, and for secure communication over local area networks or the internet.

1.3. Distributed data structures

We start with another problem from cryptography: **secret sharing**. Suppose that, for some positive integer n, we have n players that want to share a common secret in such a way that all of them together can reconstruct the secret but any subset of $n-1$ of them or less cannot do so. The reader may imagine that the secret is a key in a cryptosystem or a code guarding a common bank account or inheritance, or an authorization for a financial transaction of a company which requires the signature of a certain number of managers. This can be solved by using interpolation (Section 5.2), as follows.

We choose $2n-1$ values $f_1,\ldots,f_{n-1},u_0,\ldots,u_{n-1}$ in a field F (say, F is $\mathbb{Q}$ or a **finite field**) such that the u_i are distinct, and let f be the polynomial $f_{n-1}x^{n-1} + \cdots + f_1x + f_0$, where $f_0 \in F$ is the secret, encoded in an appropriate way. Then we give $v_i = f(u_i) = f_{n-1}u_i^{n-1} + \cdots + f_1u_i + f_0$ to player i. The reconstruction of the polynomial f from its values $v_0,\ldots,v_{n-1}$ at the n distinct points $u_0,\ldots,u_{n-1}$ is called **interpolation** and may be performed, for example, by using the **Lagrange interpolation formula** (Section 5.2). The interpolating polynomial at n points of degree less than n is unique, and hence all n players together can recover f and the secret f_0, but one can show that any proper subset of them can obtain no information on the secret. More precisely, all elements of F—as potential values of f_0—are equally consistent with the knowledge of fewer than n players. We discuss the secret sharing scheme in Section 5.3.

Essentially the same scheme works for a different problem: reliable routing. Suppose that we want to send a message consisting of several packets over a network (for example, the internet) that occasionally loses packets. We want to encode a message of length n into not many more than n packets in such a way that after a loss of up to l arbitrary packets the message can still be recovered. Such a scheme is called an **erasure code**. (We briefly discuss the related **error correcting codes**, which are designed for networks that sometimes perturb packets but do not

lose them, in Chapter 7.) An obvious solution would be to send the message $l+1$ times, but this increases message length and hence slows down communication speed by a factor of $l+1$ and is unacceptable even for small values of l.

Again we may assume that each packet is encoded as an element of some field F, and that the whole message is the sequence of packets $f_0,\ldots,f_{n-1}$. Then we choose $k=n+l$ distinct evaluation points $u_0,\ldots,u_{k-1}\in F$ and send the k packets $f(u_0),\ldots,f(u_{k-1})$ over the net. Assuming that the sequence number i is contained in the packet header and that the recipient knows $u_0,\ldots,u_{k-1}$, she can reconstruct the original message—the (coefficients of the) polynomial f—from any n of the surviving packages by interpolation (and may discard any others).

The above scheme can also be used to distribute n data blocks (for example, records of a database) among $k=n+l$ computers in such a way that after failure of up to l of them the complete information can still be recovered. The difference between secret sharing and this scheme is that in the former the relevant piece of information is only *one* coefficient of f, while in the latter it is the whole polynomial.

The above methods can be viewed as problems in **distributed data structures**. Parallel and distributed computing is an active area of research in computer science. Developing algorithms and data structures for parallel computing is a nontrivial task, often more challenging than for sequential computing. The amount of parallelism that a particular problem admits is sometimes difficult to detect. In computer algebra, **modular algorithms** (Chapters 4 and 5) provide a "natural" parallelism for a certain class of algebraic problems. These are divided into smaller problems by reduction modulo several "primes", the subproblems can be solved independently in parallel, and the solution is put together using the **Chinese Remainder Algorithm** 5.4. An important particular case is when the "primes" are linear polynomials $x-u_i$. Then modular reduction corresponds to evaluation at u_i, and the Chinese Remainder Algorithm is just interpolation at all points u_i, as in the examples above.

If the interpolation points are **roots of unity** (Section 8.2), then there is a particularly efficient method for evaluating and interpolating at those points, the **Fast Fourier Transform** (Chapters 8 and 13). It is the starting point for efficient algorithms for polynomial (and integer) arithmetic in Part II.

1.4. Computer algebra systems

We give a short overview of the computer algebra systems available at the time of writing. We do not present much detail, nor do we aim for completeness.

The vast majority of this book's material is of a fundamental nature and technology-independent. But this short section and some other places where we discuss implementations date the book; if the rapid progress in this area continues as expected, some of this material will become obsolete in a short time.

Computer algebra systems have historically evolved in several stages. An early forerunner was Williams' (1961) PMS, which could calculate floating point polynomial gcd's. The *first generation*, beginning in the late 1960s, comprised MACSYMA from Joel Moses's MATHLAB group at MIT, SCRATCHPAD from Richard Jenks at IBM, REDUCE by Tony Hearn, and SAC-I (now SACLIB) by George Collins. MUMATH by David Stoutemyer ran on a small microprocessor; its successor DERIVE is available on the hand-held TI-92. These researchers and their teams developed systems with algebraic engines capable of doing amazing *exact* (or *formal* or *symbolic*) computations: differentiation, integration, factorization, etc. The *second generation* started with MAPLE by Keith Geddes and Gaston Gonnet from the University of Waterloo in 1985 and MATHEMATICA by Stephen Wolfram. They began to provide modern interfaces and graphic capabilities, and the hype surrounding the launching of MATHEMATICA did much to make these systems widely known. The *third generation* is on the market now: AXIOM, a successor of SCRATCHPAD, by NAG, MAGMA by John Cannon at the University of Sydney, and MUPAD by Benno Fuchssteiner at the University of Paderborn. These systems incorporate a categorical approach and operator calculations.

Today's research and development of computer algebra systems is driven by three goals, which sometimes conflict: wide functionality (the capability of solving a large range of different problems), ease of use (user interface, graphics display), and speed (how big a problem you can solve with a routine calculation, say in a day on a workstation). This text will concentrate on the latter goal. We will see the basics of the fastest algorithms available today, mainly for some problems in polynomial manipulation. Several groups have developed software for these basic operations: PARI by Henri Cohen in Bordeaux, LIDIA by Johannes Buchmann in Darmstadt, NTL by Victor Shoup, and packages by Erich Kaltofen. William Stein's open-source SAGE facilitates interoperability of various systems, and SINGULAR is strong in certain algebraic settings.

A central problem is the factorization of polynomials. Enormous progress has been achieved, in particular over finite fields. In 1991, the largest problems amenable to routine calculation were about 2 KB in size, while in 1995 Shoup's software could handle problems of about 500 KB. Almost all the progress here is due to new algorithmic ideas, and this problem will serve as a guiding light for this text.

Computer algebra systems have a wide variety of applications in fields that require computations that are tedious, lengthy and difficult to get right when done by hand. In physics, computer algebra systems are used in high energy physics, for quantum electrodynamics, quantum chromodynamics, satellite orbit and rocket trajectory computations and celestial mechanics in general. As an example, Delaunay calculated the orbit of the moon under the influence of the sun and a nonspherical earth with a tilted ecliptic. This work took twenty years to complete and was published in 1867. It was shown, in 20 hours on a small computer in 1970, to be correct to nine decimal places.

Important implementation issues for a general-purpose computer algebra system concern things like the user interface (see Kajler & Soiffer 1998 for an overview), memory management and garbage collection, and which representations and simplifications are allowed for the various algebraic objects.

Their power of visualization and of solving nontrivial examples makes computer algebra systems more and more appealing for use in education. Many topics in calculus and linear algebra can be beautifully illustrated with this technology. The eleven reports in the Special Issue of the *Journal of Symbolic Computation* (Lambe 1997) give an idea of what can be achieved. The (self-referential) use in computer algebra courses is obvious; in fact, this text grew out of courses that the first author has taught at several institutions, using MAPLE (and later other systems) since 1986.

As in most active branches of science, the *sociology* of computer algebra is shaped by its leading conferences, journals, and the researchers running these. The premier research conference is the annual International Symposium on Symbolic and Algebraic Computation (ISSAC), into which several earlier streams of meetings have been amalgamated. It is run by a Steering Committee, and national societies such as the US ACM's *Special Interest Group on Symbolic and Algebraic Manipulation* (SIGSAM), the *Fachgruppe Computeralgebra* of the German GI, and others, are heavily involved. In addition, there are specialized meetings such as MEGA (multivariate polynomials), DISCO (distributed systems), and PASCO (parallel computation). Sometimes relevant results are presented at related conferences, such as STOC, FOCS, or AAECC. Several computer algebra systems vendors organize regular workshops or user group meetings.

The highly successful *Journal of Symbolic Computation*, created in 1985 by Bruno Buchberger, is the undisputed leader for research publications. Some high-quality journals with a different focus sometimes contain articles in the field: *computational complexity*, *Journal of the ACM*, *Mathematics of Computation*, and *SIAM Journal on Computing*. Other journals, such as *AAECC* and *Theoretical Computer Science*, have some overlap with computer algebra.

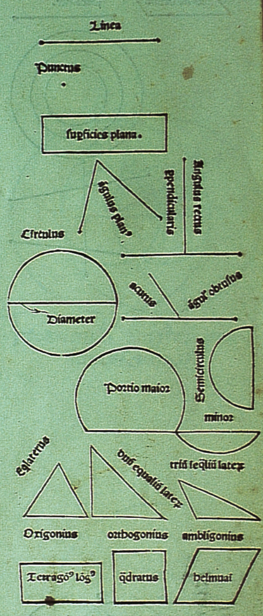
Part I

Euclid

Preclarissimus liber elementorum Euclidis perspi-
cacissimi: in artem Geometrie incipit quãfoelicissime:

Punctus est cuius ps nõ est. ¶ Linea est lõgitudo sine latitudine cui⁹ quidẽ ex-
tremitates sũt duo pũcta. ¶ Linea recta ẽ ab vno pũcto ad aliũ breuissima extẽ-
sio ĩ extremitates suas vtrũqз eoꝝ reci-
piens. ¶ Supficies ẽ q̃ lõgitudinẽ ⁊ lati-
tudinẽ tm̃ hз: cui⁹ termĩ quidẽ sũt linee.
¶ Supficies plana ẽ ab vna linea ad a-
liã extẽsio ĩ extremitates suas recipiẽs
¶ Angulus planus ẽ duarũ linearũ al-
ternus ꝯtactus: quaꝝ expãsio ẽ sup sup-
ficiẽ applicatioqз nõ directa. ¶ Quãdo aũt angulum ꝯtinẽt due
linee recte rectiline⁹ angulus noiat. ¶ Qñ recta linea sup rectã
steterit duoqз anguli vtrobiqз fuerit eq̃les: eoꝝ vterqз rect⁹ erit
¶ Lineaqз linee supstãs ei cui supstat ꝑpendicularis vocat. ¶ An-
gulus vo qui recto maior ẽ obtusus dicit. ¶ Angul⁹ vo minor re-
cto acut⁹ appellat. ¶ Termin⁹ ẽ qd vniuscuiusqз finis ẽ. ¶ Figura
ẽ q̃ tmino vl termĩs ꝯtinẽt. ¶ Circul⁹ ẽ figura plana vna q̃dem li-
nea ꝯtẽta: q̃ circũferentia noiat: in cui⁹ medio pũct⁹ ẽ: a quo oẽs
linee recte ad circũferẽtiã exeũtes sibiĩuicez sũt equales. Et hic
quidẽ pũct⁹ cẽtrũ circuli dr. ¶ Diameter circuli ẽ linea recta que
sup ei⁹ centꝝ trãsiens extremitatesqз suas circũferẽtie applicans
circulũ ĩ duo media diuidit. ¶ Semicirculus ẽ figura plana dia-
metro circuli ⁊ medietate circũferentie ꝯtenta. ¶ Portio circu-
li ẽ figura plana recta linea ⁊ parte circũferẽtie ꝯtẽta: semicircu-
lo quidẽ aut maior aut minor. ¶ Rectilinee figure sũt q̃ rectis li-
neis cõtinent quarũ quedã trilatere q̃ trib⁹ rectis lineis: quedã
quadrilatere q̃ q̃tuor rectis lineis. q̃dã mltilatere que pluribus
q̃з quatuor rectis lineis cõtinent. ¶ Figurarũ trilaterarũ: alia
est triangulus hñs tria latera equalia. Alia triangulus duo hñs
eq̃lia latera. Alia triangulus triũ inequalium laterũ. Haꝝ iterũ
alia est orthogoniũ: vnũ .s. rectum angulum habens. Alia ẽ am-
bligonium aliquem obtusum angulum habens. Alia est oxigoni
um: in qua tres anguli sunt acuti. ¶ Figurarũ autẽ quadrilateraꝝ
Alia est q̃dratum quod est equilaterũ atqз rectangulũ. Alia est
tetragon⁹ long⁹: q̃ est figura rectangula: sed equilatera non est.
Alia est helmuaym: que est equilatera: sed rectangula non est.

De principijs p se notis: ⁊ pmo de diffinitionibus earundem.



Little is known about the person of Euclid (Εὐκλείδης, c. 320–c. 275 BC). Proclus (410–485 AD) edited and commented his famous work, the *Elements* (στοιχεῖα), and summed up the meager knowledge about Euclid: *Euclid put together the* Elements, *collected many of Eudoxus' theorems, perfected many of Theaitetus', and also brought to irrefragable demonstration the things which were only somewhat loosely proved by his predecessors. This man lived in the time of the first Ptolemy. For Archimedes, who came immediately after the first (Ptolemy), makes mention of Euclid: and, further, they say that Ptolemy once asked him if there was in geometry any shorter way than that of the elements, and he answered that there was no royal road to geometry. He is then younger than the pupils of Plato but older than Eratosthenes and Archimedes; for the latter were contemporary with one another, as Eratosthenes somewhere says* (translation[1] from Heath 1925).

By interpolating between Plato's (427–347 BC) students, Archimedes (287–212 BC), and Eratosthenes (c. 275–195 BC), we can guess that Euclid lived around 300 BC; Ptolemy reigned 306–283. It is likely that Euclid learned mathematics in Athens from Plato's students. Later, he founded a school at Alexandria, and this is presumably where most of the *Elements* was written. An anecdote relates how "some one who had begun to read geometry with Euclid, when he had learned the first theorem, asked Euclid, 'But what shall I get by learning these things?' Euclid called his slave and said 'Give him threepence, since he must make gain out of what he learns.'"

In later parts of this book, we will present two giants of mathematics—Newton and Gauß—and two great mathematicians—Fermat and Hilbert. Archimedes is the third giant, in our subjective reckoning. Euclid's ticket to enter our little Hall of Fame is not a wealth of original ideas, but—besides his algorithm for the gcd—his insistence that everything must be proven from a few axioms, as he does in his systematic collection and orderly presentation of the mathematical thought of his times in the thirteen books (chapters) of his *Elements*. Probably written around 300 BC, he presents his material in the axiom-definition-lemma-theorem-proof style which—somewhat modified—has survived through the millenia. Euclid's methods go back to Aristotle's (384–322 BC) peripatetic school, and to the Eleatics.

After the Bible, the *Elements* is apparently the most often printed book, an eternal bestseller with a vastly longer half-life (before oblivion) than current-day bestsellers. It was *the* mathematical textbook for over two millenia. (In spite of their best efforts, the authors of the present text expect it to be superseded long before 4298 AD.) Even in 1908, a translator of the *Elements* exulted: *Euclid's*

[1] Reprinted with the kind permission of Dover Publications Inc., Mineola NY.

work will live long after all the text-books of the present day are superseded and forgotten. It is one of the noblest monuments of antiquity; no mathematician worthy of the name can afford not to know Euclid. Since the invention of non-Euclidean geometry and the new ideas of Klein and Hilbert in the 19th century, we don't take the *Elements* quite that seriously any longer.

In the Dark Ages, Europe's intellectuals were more interested in the maximal number of angels able to dance on a needle tip, and the *Elements* mainly survived in the Arabic civilization. The first translation from the Greek was done by Al-Ḥajjāj bin Yūsuf bin Maṭar (c. 786–835) for Caliph Hārūn al-Rashīd (766–809). These were later translated into Latin, and Erhard Ratdolt produced in Venice the first printed edition of the *Elements* in 1482; in fact, this was the first mathematics book to be printed. On page 23 we reproduce its first page from a copy in the library of the University of Basel; the underlining is possibly by the lawyer Bonifatius Amerbach, its 16th century owner, who was a friend of Erasmus.

Most of the *Elements* deals with geometry, but Books 7, 8, and 9 treat arithmetic. Proposition 2 of Book 7 asks: "Given two numbers not prime to one another, to find their greatest common measure", and the core of the algorithm goes as follows: "Let AB, CD be the two given numbers not prime to one another [...] if CD does not measure AB, then, the lesser of the numbers AB, CD being continually subtracted from the greater, some number will be left which will measure the one before it" (translation from Heath 1925).

Numbers here are represented by line segments, and the proof that the last number left (dividing the one before it) is a common divisor and the greatest one is carried out for the case of two division steps ($\ell = 2$ in Algorithm 3.6). This is Euclid's algorithm, "the oldest nontrivial algorithm that has survived to the present day" (Knuth 1998, §4.5.2), and to whose

understanding the first part of this text is devoted. In contrast to the modern version, Euclid does repeated subtraction instead of division with remainder. Since some quotient might be large, this does not give a polynomial-time algorithm, but the simple idea of removing powers of 2 whenever possible already achieves this (Exercise 3.25).

In the geometric Book 10, Euclid repeats this argument in Proposition 3 for "commensurable magnitudes", which are real numbers whose quotient is rational, and Proposition 2 states that if this process does not terminate, then the two magnitudes are incommensurable.

The other arithmetical highlight is Proposition 20 of Book 9: "Prime numbers are more than any assigned multitude of prime numbers." Hardy (1940) calls its proof "as fresh and significant as when it was discovered—two thousand years have not written a wrinkle on [it]". (For lack of notation, Euclid only illustrates his proof idea by showing how to find from three given primes a fourth one.)

It is amusing to see how after such a profound discovery comes the platitude of Proposition 21: "If as many even numbers as we please be added together, the whole is even." The *Elements* is full of such surprises, unnecessary case distinctions, and virtual repetitions. This is, to a certain extent, due to a lack of good notation. Indices came into use only in the early 19th century; a system designed by Leibniz in the 17th century did not become popular.

Euclid authored some other books, but they never hit the bestseller list, and some are forever lost.

Die ganzen Zahlen hat der liebe Gott gemacht,
alles andere ist Menschenwerk.[1]

Leopold Kronecker (1886)

"I only took the regular course." "What was that?" enquired Alice. "Reeling and Writhing, of course, to begin with," the Mock Turtle replied: "and then the different branches of Arithmetic—Ambition, Distraction, Uglification, and Derision."

Lewis Carroll (1865)

Computation is either perform'd by *Numbers*, as in Vulgar Arithmetick, or by *Species* [variables][2], as usual among Algebraists. They are both built on the same Foundations, and aim at the same End, *viz. Arithmetick* Definitely and Particularly, *Algebra* Indefinitely and Universally; so that almost all Expressions that are found out by this Computation, and particularly Conclusions, may be called *Theorems*. [...] Yet Arithmetick in all its Operations is so subservient to Algebra, as that they seem both but to make one perfect *Science of Computing*.

Isaac Newton (1728)

It is preferable to regard the computer as a handy device
for manipulating and displaying symbols.

Stanisław Marcin Ulam (1964)

In summo apud illos honore geometria fuit, itaque nihil mathematicis inlustrius; at nos metiendi ratiocinandique utilitate huius artis terminauimus modum.[3]

Marcus Tullius Cicero (45 BC)

But the rest, having no such grounds [religious devotion] of hope,
fell to another pastime, that of computation.

Robert Louis Stevenson (1889)

Check your math and the amounts entered
to make sure they are correct.

State of California (1996)

[1] God created the integers, all else is the work of man.

[2] [Text in brackets added by the authors, also in other quotations.]

[3] Among them [the Greeks] geometry was held in highest esteem, nothing was more glorious than mathematics; but we have restricted this science to the practical purposes of measuring and calculating.

2

Fundamental algorithms

We start by discussing the computer representation and fundamental arithmetic algorithms for integers and polynomials. We will keep this discussion fairly informal and avoid all the intricacies of actual computer arithmetic—that is a topic on its own. The reader must be warned that modern-day processors do *not* represent numbers and operate on them as we describe now, but to describe the tricks they use would detract us from our current goal: a simple description of how one *could*, in principle, perform basic arithmetic.

Although our straightforward approach can be improved in practice for arithmetic on small objects, say double-precision integers, it is quite appropriate for large objects, at least as a start. Much of this book deals with polynomials, and we will use some of the notions of this chapter throughout. A major goal is to find algorithmic improvements for large objects.

The algorithms in this chapter will be familiar to the reader, but she can refresh her memory of the *analysis of algorithms* with our simple examples.

2.1. Representation and addition of numbers

Algebra starts with numbers and computers work on data, so the very first issue in computer algebra is how to feed numbers as data into computers. Data are stored in pieces called **words**. Current machines use either 32- or 64-bit words; to be specific, we assume that we have a 64-bit processor. Then one machine word contains a **single precision** integer between 0 and $2^{64}-1$.

How can we represent integers outside the range $\{0,\ldots,2^{64}-1\}$? Such a **multiprecision integer** is represented by an array of 64-bit words, where the first one encodes the sign of the integer and the length of the array. To be precise, we consider the 2^{64}-ary (or radix 2^{64}) representation of a nonzero integer

$$a = (-1)^s \sum_{0\leq i\leq n} a_i \cdot 2^{64i}, \qquad (1)$$

where $s \in \{0,1\}$, $0 \leq n+1 < 2^{63}$, and $a_i \in \{0,\ldots,2^{64}-1\}$ for all i are the **digits**

(in base 2^{64}) of a. We encode it as an array

$$s \cdot 2^{63} + n + 1, a_0, \ldots, a_n$$

of 64-bit words. This representation can be made unique by requiring that the **leading digit** a_n be nonzero if $a \neq 0$ (and using the single-entry array 0 to represent $a = 0$). We will call this the **standard representation** for a. For example, the standard representation of -1 is $2^{63} + 1, 1$. It is, however, convenient also to allow nonstandard representations with leading zero digits since this sometimes facilitates memory management, but we do not want to go into details here. The range of integers that can be represented in standard representation on a 64-bit processor is between $-2^{64 \cdot 2^{63}} + 1$ and $2^{64 \cdot 2^{63}} - 1$; each of the two boundaries requires $2^{63} + 1$ words of storage. This size limitation is quite sufficient for practical purposes: one of the larger representable numbers would fill about 70 million 1-TB-discs.

For a nonzero integer $a \in \mathbb{Z}$, we define the **length** $\lambda(a)$ of a as

$$\lambda(a) = \lfloor \log_{2^{64}} |a| \rfloor + 1 = \left\lfloor \frac{\log_2 |a|}{64} \right\rfloor + 1,$$

where $\lfloor \cdot \rfloor$ denotes rounding down to the nearest integer (so that $\lfloor 2.7 \rfloor = 2$ and $\lfloor -2.7 \rfloor = -3$). Thus $\lambda(a) + 1 = n + 2$ is the number of words in the standard representation (1) of a (see Exercise 2.1). This is quite a cluttered expression, and it is usually sufficient to know that about $\frac{1}{64} \log_2 |a|$ words are needed, or even more succinctly $O(\log_2 |a|)$, where the **big-Oh notation** "O" hides an arbitrary constant (Section 25.7).

We assume that our hypothetical processor has at its disposal a command for the addition of two single precision integers a and b. The output of the addition command is a 64-bit word c plus the content of the **carry flag** $\gamma \in \{0, 1\}$, a special bit in the processor status word which indicates whether the result exceeds 2^{64} or not. In order to be able to perform addition of multiprecision integers more easily, the carry flag is also *input* to the addition command. More precisely, we have

$$a + b + \gamma = \gamma^* \cdot 2^{64} + c,$$

where γ is the value of the carry flag before the addition and γ^* is its value afterwards. Usually there are processor instructions to clear and set the carry flag.

If $a = \sum_{0 \le i \le n} a_i 2^{64i}$ and $b = \sum_{0 \le i \le m} b_i 2^{64i}$ are two multiprecision integers, then their sum is

$$c = \sum_{0 \le i \le k} (a_i + b_i) 2^{64i},$$

where $k = \max\{n, m\}$, and if, say, $m \le n$, then $b_{m+1}, \ldots, b_n$ are set to zero. (In other words, we may assume that $m = n$.) In general, $a_i + b_i$ may be larger than 2^{64}, and if so, then the carry has to be added to the next digit in order to get a 2^{64}-ary

representation again. This process propagates from the lower order to the higher order digits, and in the worst case, a carry from the addition of a_0 and b_0 may influence the addition of a_n and b_n, as the example $a = 2^{64(n+1)} - 1$ and $b = 1$ shows. Here is an algorithm for the addition of two multiprecision integers of the same sign; see Exercise 2.3 for a subtraction algorithm.

ALGORITHM 2.1 Addition of multiprecision integers.

Input: Multiprecision integers $a = (-1)^s \sum_{0 \le i \le n} a_i 2^{64i}$, $b = (-1)^s \sum_{0 \le i \le n} b_i 2^{64i}$, not necessarily in standard representation, with $s \in \{0, 1\}$.

Output: The multiprecision integer $c = (-1)^s \sum_{0 \le i \le n+1} c_i 2^{64i}$ such that $c = a + b$.

1. $\gamma_0 \longleftarrow 0$
2. **for** $i = 0, \ldots, n$ **do**
   $c_i \longleftarrow a_i + b_i + \gamma_i, \quad \gamma_{i+1} \longleftarrow 0$
   if $c_i \ge 2^{64}$ **then** $c_i \longleftarrow c_i - 2^{64}, \quad \gamma_{i+1} \longleftarrow 1$
3. $c_{n+1} \longleftarrow \gamma_{n+1}$
   return $(-1)^s \sum\limits_{0 \le i \le n+1} c_i 2^{64i}$

We use the sign $\longleftarrow$ in algorithms to denote an assignment, and indentation distinguishes the body of a loop.

We practise this algorithm on the addition of $a = 9438 = 9 \cdot 10^3 + 4 \cdot 10^2 + 3 \cdot 10 + 8$ and $b = 945 = 9 \cdot 10^2 + 4 \cdot 10 + 5$ and in decimal (instead of 2^{64}-ary) representation:

i	4	3	2	1	0
a_i		9	4	3	8
b_i		0	9	4	5
γ_i	1	1	0	1	0
c_i	1	0	3	8	3

How much time does this algorithm use? The basic subroutine, the addition of two single precision integers, requires some number of machine cycles, say k. This number will depend on the processor used, and the actual CPU time depends on the processor's speed. Thus the addition of two integers of length at most n takes kn machine cycles for single precision additions, plus some number of cycles for control structures, manipulating sign bits, index arithmetic, memory access, etc. We will, however, (correctly) assume that the number of machine cycles for the latter has the same order of magnitude as the cost for the single precision arithmetic operations that we count.

For the remainder of this text, it is vital to abstract from machine-dependent details as discussed above. We will then just say that the addition of two n-word

integers can be done in time $O(n)$, or at cost $O(n)$, or with $O(n)$ **word operations**; the constants hidden in the big-Oh will depend on the details of the machine. We gain two advantages from this concept: a shorter and more intuitive notation, and independence of particular machines. The abstraction is justified by the fact that the *actual* performance of an algorithm often depends on compiler optimization, clever cache usage, pipelining effects, and many other things that are quite technical and nearly impossible to describe in a comparatively high-level programming language. However, experiments show that "big-Oh" statements are reflected surprisingly well by implementations on any kind of sequentially working processor: adding two multiprecision integers is a linear operation in the sense that doubling the input size also approximately doubles the running time.

One can make these statements more precise and formally satisfying. The cost measure that is widely used for algorithms dealing with integers is the number of **bit operations** which can be rigorously defined as the number of steps of a Turing or register machine (random access machine, RAM) or the number of gates of a Boolean circuit implementing the algorithm. Since the details of those computational models are rather technical, however, we will content ourselves with informal arguments and cost measures, as above.

Related data types occurring in currently available processors and mathematical software are single and multiprecision **floating point numbers**. These represent approximations of real numbers, and arithmetic operations, such as addition and multiplication, are subject to **rounding errors**, in contrast to the arithmetic operations on multiprecision integers, which are **exact**. Algorithms based on computations with floating point numbers are the main topic in *numerical analysis*, which is a theme of its own; neither it nor the recent attempts at systematically combining exact and numerical computations will be discussed in this text.

2.2. Representation and addition of polynomials

The two main data types on which our algorithms operate are **numbers** as above and **polynomials**, such as $a = 5x^3 - 4x + 3 \in \mathbb{Z}[x]$. In general, we have a commutative **ring** R, such as $\mathbb{Z}$, in which we can perform the operations of addition, subtraction, and multiplication according to the usual rules; see Section 25.2 for details. (All our rings have a multiplicative unit element 1.) If we can also divide by any nonzero element, as in the rational numbers $\mathbb{Q}$, then R is a **field**.

A polynomial $a \in R[x]$ in x over R is a finite sequence $(a_0, \ldots, a_n)$ of elements of R (the **coefficients** of a), for some $n \in \mathbb{N}$, and we write it as

$$a = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = \sum_{0 \le i \le n} a_i x^i. \tag{2}$$

If $a_n \neq 0$, then $n = \deg a$ is the **degree** of a, and $a_n = \operatorname{lc}(a)$ is its **leading coefficient**. If $\operatorname{lc}(a) = 1$, then a is **monic**. It is convenient to take $-\infty$ as the degree of the zero

polynomial. We can represent a by an array whose ith element is a_i (in analogy to the integer case, we would also need some storage for the degree, but we will neglect this). This assumes that we already have a way of representing coefficients from R. The length (the number of ring elements) of this representation is $n+1$.

For an integer $r \in \mathbb{N}_{>1}$ (in particular, for $r = 2^{64}$ as in the previous section), the representations (2) of a polynomial and the radix r representation

$$a = a_n r^n + a_{n-1} r^{n-1} + \cdots + a_1 r + a_0 = \sum_{0 \le i \le n} a_i r^i,$$

with digits $a_0, \ldots, a_n \in \{0, \ldots, r-1\}$, of an integer a are quite similar. This is particularly visible if we take polynomials over $R = \mathbb{Z}_r = \{0, \ldots, r-1\}$, the ring of integers modulo r, with addition and multiplication modulo r (Sections 4.1 and 25.2). This similarity is an important point for computer algebra; many of our algorithms apply (with small modifications) to both the integer and polynomial cases: multiplication, division with remainder, gcd and Chinese remainder computation. It is also relevant to note where this does not apply: the subresultant theory (Chapter 6) and, most importantly, the factorization problem (Parts III and IV). At the heart of this distinction lies the deceptively simple carry rule. It gives the low digits some influence on the high digits in addition of integers, and messes up the cleanly separated rules in the addition of two polynomials

$$a = \sum_{0 \le i \le n} a_i x^i \text{ and } b = \sum_{0 \le i \le m} b_i x^i \tag{3}$$

in $R[x]$. This is quite easy:

$$c = a + b = \sum_{0 \le i \le n} (a_i + b_i) x^i = \sum_{0 \le i \le n} c_i x^i,$$

where the addition $c_i = a_i + b_i$ is performed in R and, as with integers, we may assume that $m = n$.

For example, addition of the polynomials $a = 9x^3 + 4x^2 + 3x + 8$ and $b = 9x^2 + 4x + 5$ in $\mathbb{Z}[x]$ works as follows:

i	3	2	1	0
a_i	9	4	3	8
b_i	0	9	4	5
c_i	9	13	7	13

Here is the (rather trivial) algorithm in our formalism.

ALGORITHM 2.2 Addition of polynomials.

Input: $a = \sum_{0 \le i \le n} a_i x^i$, $b = \sum_{0 \le i \le n} b_i x^i$ in $R[x]$, where R is a ring.
Output: The coefficients of $c = a + b \in R[x]$.

1. **for** $i = 0, \ldots, n$ **do** $c_i \longleftarrow a_i + b_i$
2. **return** $c = \sum_{0 \le i \le n} c_i x^i$

It is somewhat simpler than integer addition, with its carries. This simplicity propagates down the line for more complicated algorithms such as multiplication, division with remainder, etc. Although integers are more intuitive (we learn about them at a much earlier stage in life), their algorithms are a bit more involved, and we adopt in this book as a general program the strategy to present mainly the simpler polynomial case which allows us to concentrate on the essentials, and often leave details in the integer case to the exercises.

As a first example, we have seen that addition of two polynomials of degree up to n takes at most $n+1$ or $O(n)$ arithmetic operations in R; there is no concern with machine details here. This is a much coarser cost measure than the number of word operations for integers. If, for example, $R = \mathbb{Z}$ and the coefficients are less than B in absolute value, then the cost in word operations is $O(n \log B)$, which is the same order of magnitude as the input size. Moreover, additive operations $+, -$ in R are counted at the same cost as multiplicative operations $\cdot, /$, while in most applications the latter are significantly more expensive than the former.

As a general rule, we will analyze the number of **arithmetic operations** in the ring R (additions and multiplications, and also divisions if R is a field) used by an algorithm. In our analyses, the word *addition* stands for *addition or subtraction*; we do not count the latter separately. The number of other operations, such as index calculations or memory accesses, tends to be of the same order of magnitude. These are usually performed with machine instructions on single words, and their cost is negligible when the arithmetic quantities are large, say multiprecision integers. The input size is the number of ring elements that the input occupies. If the coefficients are integers or polynomials themselves, we may then consider separately the size of the coefficients involved and the cost for coefficient arithmetic.

We try to provide explicit (but not necessarily minimal) constants for the dominant term in our analyses of algorithms on polynomials when the cost measure is the number of arithmetic operations in the coefficient ring, but confine ourselves to O-estimates when counting the number of word operations for algorithms working on integers or polynomials with integral coefficients.

2.3. Multiplication

Following our program, we first consider the product $c = a \cdot b = \sum_{0 \le k \le n+m} c_k x^k$ of two polynomials a and b in $R[x]$, as in (3). Its coefficients are

$$c_k = \sum_{\substack{0 \le i \le n \\ 0 \le j \le m \\ i+j=k}} a_i b_j \tag{4}$$

for $0 \le k \le n+m$. We can just take this formula and turn it into a subroutine, after figuring out suitable loop variables and boundaries:

> **for** $k = 0, \ldots, n+m$ **do**
> $c_k \longleftarrow 0$
>   **for** $i = \max\{0, k-m\}, \ldots, \min\{n,k\}$ **do**
> $c_k \longleftarrow c_k + a_i \cdot b_{k-i}$

There are other ways to organize the loops. We learned in school the following algorithm.

ALGORITHM 2.3 Multiplication of polynomials.

Input: The coefficients of $a = \sum_{0 \le i \le n} a_i x^i$ and $b = \sum_{0 \le i \le m} b_i x^i$ in $R[x]$, where R is a (commutative) ring.

Output: The coefficients of $c = a \cdot b \in R[x]$.

1. **for** $i = 0, \ldots, n$ **do** $d_i \longleftarrow a_i x^i \cdot b$
2. **return** $c = \sum_{0 \le i \le n} d_i$

The multiplication $a_i x^i \cdot b$ is realized as the multiplication of each b_j by a_i plus a shift by i places. The variable x serves us just as a convenient way to write polynomials, and there is no arithmetic involved in "multiplying" by x or any power of it. Here is a small example:

$$\begin{array}{rrrrrrr}
5x^2+2x+1\cdot & & & 2x^3 & +x^2 & +3x & +5 \\ \hline
 & & & 2x^3 & +x^2 & +3x & +5 \\
 & & +4x^4 & +2x^3 & +6x^2 & +10x & \\
 & +10x^5 & +5x^4 & +15x^3 & +25x^2 & & \\ \hline
 & 10x^5 & +9x^4 & +19x^3 & +32x^2 & +13x & +5
\end{array}$$

How much time does this take, that is, how many operations in the ground ring R? Each of the $n+1$ coefficients of a has to be multiplied with each of the $m+1$ coefficients of b, for a total of $(n+1)(m+1)$ multiplications. Then these are summed up in $n+m+1$ sums; summing s items costs $s-1$ additions. So the total number of additions is

$$(n+1)(m+1) - (n+m+1) = nm,$$

and the total cost for multiplication is $2nm+n+m+1 \le 2(n+1)(m+1)$ operations in R. (If a is monic, then the bound drops to $2nm+n \le 2n(m+1)$.) Thus we can say that two polynomials of degree at most n can be multiplied using $2n^2+2n+1$ operations, or $2n^2+O(n)$ operations, or $O(n^2)$ operations,

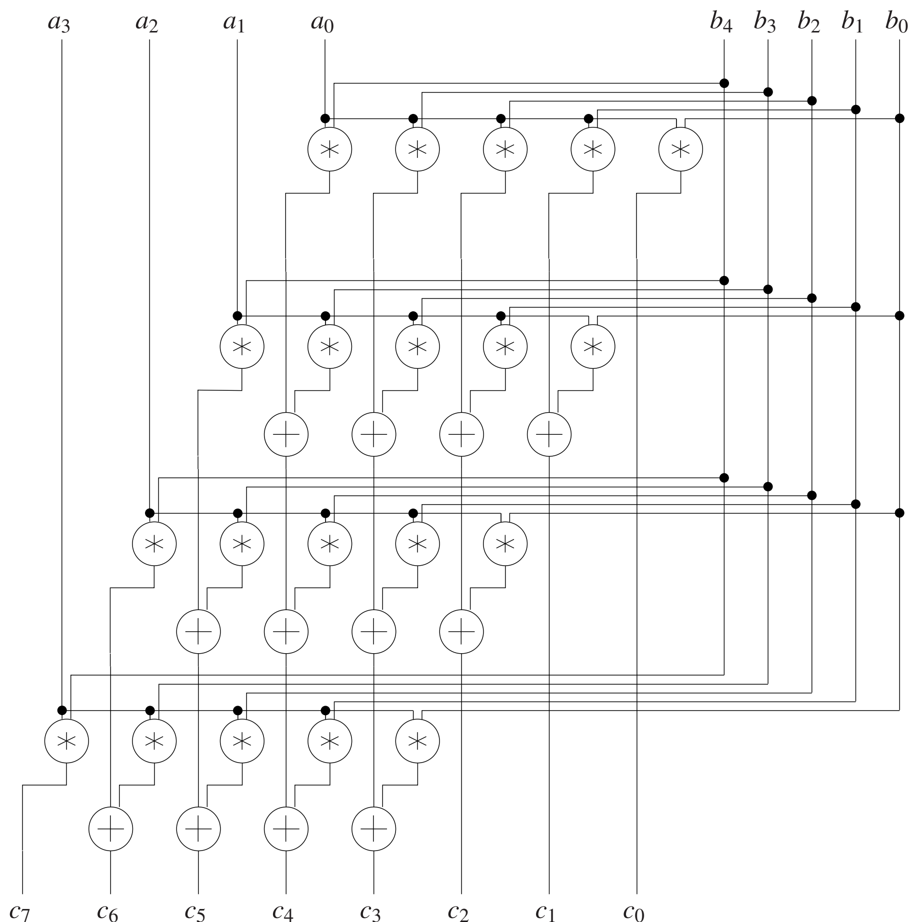


FIGURE 2.1: An arithmetic circuit for polynomial multiplication. The flow of control is directed downwards. An "electrical" view is to think of the edges as lines, where ring elements "flow", with "contact" crossings marked with a •, and no contact at the other crossings. The size of this circuit equals 32, the number of arithmetic gates in it.

or in **quadratic time**. The three expressions for the running time get progressively simpler but also less precise. In this book, each of the three versions has its place (and there are even more versions).

For a computer implementation, Algorithm 2.3 has the drawback of requiring us to store $n+1$ polynomials with $m+1$ coefficients each. A way around this is to interleave the final addition with the computation of the $a_i x^i b$. This takes the same time but uses only $O(n+m)$ storage and is shown in Figure 2.1 for $n=3$ and $m=4$. Each horizontal level corresponds to one pass through the loop body in step 1.

We will call **classical** those algorithms that take a definition of a function and implement it fairly literally, as the multiplication algorithms above implements the

formula (4). One might think that this is the only way of doing it. Fortunately, there are much faster ways of multiplying, in almost linear rather than quadratic time. We will study these fast algorithms in Part II. By contrast, for the addition problem no improvement is possible, nor is it necessary: the algorithm uses only linear time.

According to our general program, we now examine the integer case. The product of two single precision integers a, b between 0 and $2^{64}-1$ has "double precision": it lies in the interval $\{0, \ldots, 2^{128}-2^{65}+1\}$. We assume that our processor has a single precision multiplication instruction which returns the product in two 64-bit words c, d such that $a \cdot b = d \cdot 2^{64} + c$. Here is the integer analog of Algorithm 2.3.

ALGORITHM 2.4 Multiplication of multiprecision integers.

Input: Multiprecision integers $a = (-1)^s \sum_{0 \leq i \leq n} a_i 2^{64i}$, $b = (-1)^t \sum_{0 \leq i \leq m} b_i 2^{64i}$, not necessarily in standard representation, with $s, t \in \{0, 1\}$.

Output: The multiprecision integer ab.

1. **for** $i = 0, \ldots, n$ **do** $d_i \longleftarrow a_i 2^{64i} \cdot |b|$
2. **return** $c = (-1)^{s+t} \sum_{0 \leq i \leq n} d_i$

Besides the multiplication by 2^{64i}, which is just a shift in the 2^{64}-ary representation, multiplication of a multiprecision integer b by a single precision integer a_i must be implemented. The time for this is $O(m)$ (Exercise 2.5), and the total time is quadratic: $O(nm)$. In line with our general program, we omit the details for implementing this efficiently.

We conclude this section with the example multiplication of $a = 521 = 5 \cdot 10^2 + 2 \cdot 10 + 1$ and $b = 2135 = 2 \cdot 10^3 + 10^2 + 3 \cdot 10 + 5$ in decimal representation, according to Algorithm 2.4.

$$\begin{array}{rr} 521 \cdot & 2135 \\ \hline & 2135 \\ & +42700 \\ & +1067500 \\ \hline & 1112335 \end{array}$$

2.4. Division with remainder

In many applications, calculations "modulo an integer" play an important role; examples from program checking, database integrity, coding theory and cryptography are discussed later in the book. But even when one is really only interested in integer problems, a "modular" approach is often computationally successful; we will see this for gcds and factorization of integer polynomials.

The basic tool for modular arithmetic is **division with remainder**: given integers a, b, with b nonzero, we want to find a **quotient** q and a **remainder** r—both integers—so that

$$a = qb + r, \quad |r| < |b|.$$

In line with our general program, we first discuss the computational aspect of this problem for polynomials. So we are given $a, b \in R[x]$, with b nonzero, and want to find $q, r \in R[x]$ so that

$$a = qb + r, \quad \deg r < \deg b. \tag{5}$$

A first problem is that such q and r do not always exist: it is impossible to divide x^2 by $2x + 1$ with remainder in $\mathbb{Z}[x]$! (See Exercise 2.8.) There is a way around this, the **pseudodivision** explained in Section 6.12. However, for the moment we simplify the problem by assuming that the leading coefficient $\mathrm{lc}(b)$ of b is a **unit** in R, so that it has an inverse $v \in R$ with $\mathrm{lc}(b)v = 1$. For $R = \mathbb{Z}$, that still only allows 1 or -1 as leading coefficient, but when R is a field, division with remainder by an arbitrary nonzero polynomial is possible.

We remind the reader of the "synthetic division" learned in high school with a small example in $\mathbb{Z}[x]$:

$$\begin{array}{rrrrl}
3x^4 & +2x^3 & & +x+5 & : x^2+2x+3 = 3x^2-4x-1 \\
-3x^4 & -6x^3 & -9x^2 & & \\
\hline
 & -4x^3 & -9x^2 & +x+5 & \\
 & +4x^3 & +8x^2 & +12x & \\
\hline
 & & -x^2 & +13x+5 & \\
 & & +x^2 & +2x+3 & \\
\hline
 & & & 15x+8 &
\end{array}$$

Thus the coefficients of the quotient $q = 3x^2 - 4x - 1$ are determined one by one, starting at the top, by setting them equal to the corresponding coefficient of the current "remainder" (in general, one additionally has to divide by $\mathrm{lc}(b)$), which initially is $a = 3x^4 + 2x^3 + x + 5$. Then the remainder is adjusted by subtracting the appropriate multiple of $b = x^2 + 2x + 3$. The final remainder is $r = 15x + 8$. The degree of q is $\deg a - \deg b$ if $q \neq 0$. The following algorithm formalizes this familiar *classical method* for division with remainder by a polynomial whose leading coefficient is a unit.

ALGORITHM 2.5 Polynomial division with remainder.

Input: $a = \sum_{0 \le i \le n} a_i x^i, b = \sum_{0 \le i \le m} b_i x^i \in R[x]$, with all $a_i, b_i \in R$, where R is a ring (commutative, with 1), b_m a unit, and $n \ge m \ge 0$.

Output: $q, r \in R[x]$ with $a = qb + r$ and $\deg r < m$.

1. $r \longleftarrow a, \quad u \longleftarrow b_m^{-1}$

2. **for** $i = n-m, n-m-1, \ldots, 0$ **do**
3. **if** $\deg r = m+i$ **then** $q_i \longleftarrow \mathrm{lc}(r)u, \quad r \longleftarrow r - q_i x^i b$
   else $q_i \longleftarrow 0$
4. **return** $q = \sum_{0 \le i \le n-m} q_i x^i$ and r

Figure 2.2 represents this algorithm for $n = 7$ and $m = 4$ when $b_m = 1$. Each horizontal level in the circuit corresponds to one pass through step 3.

As in the polynomial multiplication algorithm, the multiplication $q_i x^i b$ in step 3 is just a multiplication of each b_j by q_i, followed by a shift by i places. The $(m+i)$th coefficient of r becomes 0 in step 3 and hence need not be computed. Thus the cost for one execution of step 3 is $m+1$ multiplications and m additions

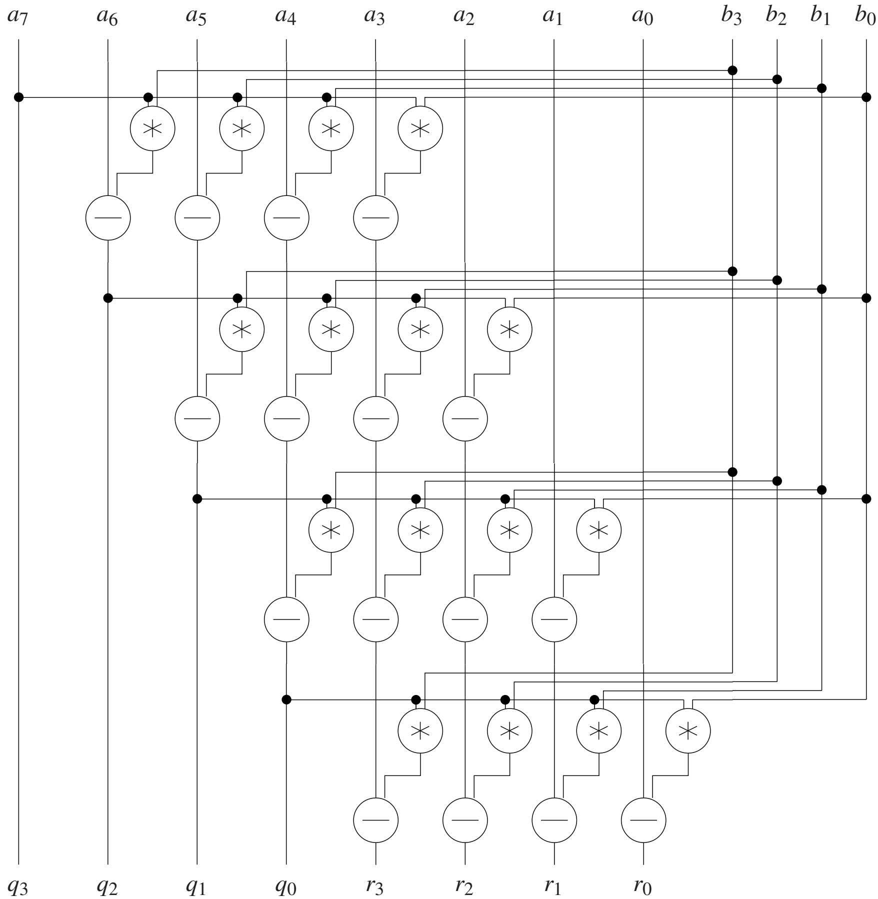


FIGURE 2.2: An arithmetic circuit for polynomial division. A subtraction node computes the difference of its left input minus its right input.

in R if $\deg r = m+i$. Together, we have a cost of at most

$$(2m+1)(n-m+1) = (2\deg b+1)(\deg q+1) \in O(n^2)$$

additions and multiplications in R plus one division for inverting b_m, and only at most $2\deg b(\deg q+1)$ additions and multiplications if b is monic. In many applications, we have $n < 2m$, and then the cost is at most $2m^2 + O(m)$ ring operations (plus an inversion), which is essentially the same as for multiplying two polynomials of degree at most m.

It is easy to see that the quotient and remainder are uniquely determined (when $\mathrm{lc}(b)$ is a unit). Namely, another equation $a = q^*b + r^*$, with $q^*, r^* \in R[x]$ and $\deg r^* < \deg b$, yields by subtraction

$$(q^* - q)b = r - r^*.$$

The right hand side has degree less than $\deg b$, and the left hand side has degree at least $\deg b$, unless $q^* - q = 0$. Therefore the latter is true, $q = q^*$, and $r = r^*$. We write "a quo b" for the quotient q and "a rem b" for the remainder r.

What about the integer case? The analog of Algorithm 2.5 is well known from high school, at least in the decimal representation:

$$\begin{array}{r l} 32015 & : 123 = 260 \\ -24600 & \\ \hline 7415 & \\ -7380 & \\ \hline 35 & \end{array}$$

The digits of the quotient are again determined one by one, starting at the top. At each step, one has to determine how often the leading digit of b divides the one or two leading digits of the current remainder. This requires a "double-by-single-precision" division instruction. However, it may happen that this "trial" quotient digit is too large, as in the first line: the quotient of 3 on division by 1 is 3, but the leading digit of the quotient $\lfloor 32015/123 \rfloor$ is 2.

Algorithmically, in each step, the product of the shifted divisor $2^{64i}b$ with a trial quotient digit is subtracted from some integer of the same length. If the result is negative, then the trial quotient digit is too large and one has to decrement it and add $2^{64i}b$ to the remainder (repeatedly, if necessary) until the remainder is nonnegative. If things are arranged properly, then one such step can be done with $O(\lambda(b))$ word operations. The length of the quotient—the number of iterations—is at most $\lambda(a) - \lambda(b) + 1$ (Exercise 2.7), and the overall cost estimate is again $O(\lambda(b)\lambda(q))$ or $O(m(n-m))$ word operations if a and b have lengths n and m, respectively. Due to the carries, the details are somewhat more complicated than in the polynomial case; the interested reader is referred to the comprehensive discussion in Knuth (1998), §4.3.1.

In contrast to the polynomial case, the remainder r is not uniquely determined by the condition $|r| < |b|$: $13 = 2 \cdot 5 + 3 = 3 \cdot 5 + (-2)$. We will often follow the convention that the remainder be nonnegative and denote it by a rem b, and the corresponding quotient then is $\lfloor a/b \rfloor$ if $b > 0$.

Notes. Good texts on algorithms and their analysis are Brassard & Bratley (1996) and Cormen, Leiserson, Rivest & Stein (2009).

Addition and multiplication algorithms in decimal notation are explicitly described in Stevin (1585). Several algorithms for computer arithmetic, such as fast (carry look-ahead and carry-save) addition, are given in Cormen, Leiserson, Rivest & Stein (2009). For information about the highly active area of symbolic-numeric computations see the Special Issue of the *Journal of Symbolic Computation* (Watt & Stetter 1998) and Corless, Kaltofen & Watt (2003).

2.4. The first algorithm for division with remainder of polynomials appears in Nuñez (1567). He is, of course, limited by the concepts of his times to specific degrees, 3 and 1 in his case, and positive coefficients. On f°31r°, Nuñez writes: *Si el partidor fuere compuesto, partiremos las mayores dignidades de lo que se ha de partir por la mayor dignidad del partidor, dexandole en que pueda caber la otra dignidad del partidor, y lo q̃ viniere multiplicaremos por el partidor, y lo produzido por essa multiplicacion sacaremos de toda la sũma que se parte, y lo mismo obraremos en lo q̃ restare, por el modo q̃ tenemos quando partimos numero por numero. Y llegando a numero o dignidad en esta obra que sea de menor denominacion, que el partidor, quedara essa quantidad en quebrado, [...]*[1] He then explains the division of $12x^3 + 18x^2 + 27x + 17$ by $4x + 3$ with quotient $3x^2 + 2\frac{3}{4}x + 5\frac{3}{16}$ and remainder $1\frac{13}{16}$, and checks his result by multiplying out.

An anonymous author (1835) presents a decimal division algorithm for hand calculation based on a "10's complement" notation.

Exercises.

2.1 For an integer $r \in \mathbb{N}_{>1}$, we consider the variable-length radix r representation $(a_0, \ldots, a_{l-1})$ of a positive integer a, with $a = \sum_{0 \le i < l} a_i r^i$, $a_0, \ldots, a_{l-1} \in \{0, \ldots, r-1\}$, and $a_{l-1} \neq 0$. Prove that its length l is $\lfloor \log_r a \rfloor + 1$.

2.2 Design a representation for integers of unlimited size on a 64-bit machine.

2.3 (i) Specify a processor instruction analogous to the addition instruction mentioned in the text which performs subtraction of two single precision integers. Use the carry flag to indicate whether the result is negative or not.

(ii) Design an algorithm similar to Algorithm 2.1 for the subtraction of two multiprecision integers a and b of equal sign and with $|a| > |b|$.

(iii) Discuss how to decide whether $|a| > |b|$ holds.

2.4 Here is a piece of code implementing Algorithm 2.1 for nonnegative multiprecision integers (that is, when $s = 0$) on a hypothetical processor. Text enclosed in `/*` and `*/` is a comment. The

[1] If the divisor is composed [of more than one summand], we divide the leading term of the dividend by the leading term of the divisor, ignoring the other terms of the divisor, and we multiply the result by the divisor and subtract the result of this multiplication from the whole of the dividend, and we apply the same procedure to what is left, in the way we use it when we divide one number by another. And if we arrive in this procedure at numbers or terms whose degree is less than that of the divisor, then this quantity will remain as a fraction [...]

processor has 26 freely usable registers named A to Z. Initially, registers A and B point to the first word (the one containing the length) of the representations of a and b, respectively, and C points to a piece of memory where the representation of c shall be placed.

```
 1: LOAD N, [A]    /* load the word that A points to into register N */
 2: ADD K, N, 1    /* add 1 to register N and store the result in K
                      (without affecting the carry flag) */
 3: STORE [C], K   /* store K in the word that C points to */
 4: ADD A, A, 1    /* increase register A by 1 */
 5: ADD B, B, 1
 6: ADD C, C, 1
 7: LOAD I, 1      /* load the constant 1 into register I */
 8: CLEARC         /* clear carry flag */
 9: COMP I, N      /* compare the contents of registers I and N ...  */
10: BGT 20         /* ...  and jump to line 20 if I is greater */
11: LOAD S, [A]
12: LOAD T, [B]
13: ADDC S, S, T   /* add the contents of register T to register S
                      using the carry flag */
14: STORE [C], S
15: ADD A, A, 1
16: ADD B, B, 1
17: ADD C, C, 1
18: ADD I, I, 1
19: JMP 9          /* unconditionally jump to line 9 */
20: ADDC S, 0, 0   /* store carry flag in S */
21: STORE [C], S
22: RETURN
```

Suppose that our processor runs at 2 GHz and that the execution of one instruction takes one machine cycle = 0.5 nanoseconds $= 5 \cdot 10^{-10}$ seconds. Calculate the precise time, in terms of n, to run the above piece of code, and convince yourself that this is indeed $O(n)$.

2.5 Give an algorithm for multiplying a multiprecision integer b by a single precision integer a, making use of the single precision multiply instruction described in Section 2.3. Show that your algorithm uses $\lambda(b)$ single precision multiplications and the same number of single precision additions. Convert your algorithm into a machine program as in Exercise 2.4.

2.6 Prove that $\max\{\lambda(a), \lambda(b)\} \le \lambda(a+b) \le \max\{\lambda(a), \lambda(b)\} + 1$ and $\lambda(a) + \lambda(b) - 1 \le \lambda(ab) \le \lambda(a) + \lambda(b)$ hold for all $a, b \in \mathbb{N}_{>0}$.

2.7 Let $a > b \in \mathbb{N}_{>0}$, $m = \lambda(a)$, $n = \lambda(b)$ and $q = \lfloor a/b \rfloor$. Give tight upper and lower bounds for $\lambda(q)$ in terms of m and n.

2.8 Prove that in $\mathbb{Z}[x]$ one cannot divide x^2 by $2x+1$ with remainder as in (5).

2.9* Let R be an integral domain with field of fractions K and $a, b \in R[x]$ of degree $n \ge m \ge 0$. Then we can apply the polynomial division algorithm 2.5 to compute $q, r \in K[x]$ such that $a = qb + r$ and $\deg r < \deg b$.

(i) Prove that there exist $q, r \in R[x]$ with $a = qb + r$ and $\deg r < \deg b$ if and only if $\mathrm{lc}(b) \mid \mathrm{lc}(r)$ in R every time the algorithm passes through step 3, and that they are unique in that case.

(ii) Modify Algorithm 2.5 so that on input a, b, it decides whether $q, r \in R[x]$ as in (i) exist, and if so, computes them. Show that this takes the same number of operations in R as given in the text, where one operation is either an addition or a multiplication in R, or a test which decides whether an element $c \in R$ divides another element $d \in R$, and if so, computes the quotient $d/c \in R$.

2.10 Let R be a ring (commutative, with 1) and $a = \sum_{0 \le i \le n} a_i x^i \in R[x]$ of degree n, with all $a_i \in R$. The **weight** $w(a)$ of a is the number of nonzero coefficients of a besides the leading coefficient:

$$w(a) = \#\{0 \le i < n : a_i \ne 0\}.$$

Thus $w(a) \le \deg a$, with equality if and only if all coefficients of a are nonzero. The **sparse** representation of a, which is particularly useful if a has small weight, is a list of pairs $(i, a_i)_{i \in I}$, with each $a_i \in R$ and $a = \sum_{i \in I} a_i x^i$. Then we can choose $\#I = w(a) + 1$.

(i) Show that two polynomials $a, b \in R[x]$ of weight $n = w(a)$ and $m = w(b)$ can be multiplied in the sparse representation using at most $2nm + n + m + 1$ arithmetic operations in R.

(ii) Draw an arithmetic circuit for division of a polynomial $a \in R[x]$ of degree less than 9 by $b = x^6 - 3x^4 + 2$ with remainder. Try to get its size as small as possible.

(iii) Let $n \ge m$. Show that quotient and remainder on division of a polynomial $a \in R[x]$ of degree less than n by $b \in R[x]$ of degree m, with $\mathrm{lc}(b)$ a unit, can be computed using $n - m$ divisions in R, and $w(b) \cdot (n - m)$ multiplications and subtractions in R each.

2.11 Let R be a ring and $k, m, n \in \mathbb{N}$. Show that the "classical" multiplication of two matrices $A \in R^{k \times m}$ and $B \in R^{m \times n}$ takes $(2m - 1)kn$ arithmetic operations in R.

‘Immortality’ may be a silly word, but probably a mathematician
has the best chance of whatever it may mean.

Godfrey Harold Hardy (1940)

The ignoraunte multitude doeth, but as it was euer wonte, enuie that
knoweledge, whiche thei can not attaine, and wishe all men ignoraunt,
like unto themself. [. . .] Yea, the *pointe* in *Geometrie*,
and the unitie in *Arithmetike*, though bothe be undiuisible,
doe make greater woorkes, & increase greater multitudes,
then the brutishe bande of ignoraunce is hable to withstande.

Robert Recorde (1557)

If mathematics is considered to be a science
[that is, devoted to the description of nature and its laws],
it is more fundamental than any other.

Murray Gell-Mann (1994)

I have often wished, that I had employed about the speculative part of
geometry, and the cultivation of the specious Algebra [multivariate
polynomials] I had been taught very young, a good part of that time
and industry, that I had spent about surveying and fortification (of
which I remember I once wrote an entire treatise) and other practick
parts of mathematicks. And indeed the operations of symbolical
arithmetick (or the modern Algebra) seem to me to afford men one of
the clearest exercises of reason that I ever yet met with.

Robert Boyle (1671)

The length of this article will not be blamed by any one
who considers that, the sacred writers excepted, no Greek
has been so much read and so variously translated as Euclid.

Augustus De Morgan (c. 1844)

3
The Euclidean Algorithm

Integers and polynomials with coefficients in a field behave similarly in many respects. Often—but not always—the algorithms for both types of objects are quite similar, and sometimes one can find a common abstraction of both domains, and it is then sufficient to design one algorithm for this generalization to solve both problems in one fell swoop. In this chapter, the Euclidean domain covers the structural similarities between gcd computations for integers and polynomials. Typically, in such a situation the polynomial version is slightly simpler, and in Chapter 6, we will meet polynomial subresultants which have no integer analog at all.

3.1. Euclidean domains

The Euclidean Algorithm for the two integers 126 and 35 works as follows:

$$\begin{array}{rcl} 126 & = & 3\cdot 35+21, \\ 35 & = & 1\cdot 21+14, \\ 21 & = & 1\cdot 14+7, \\ 14 & = & 2\cdot 7, \end{array} \tag{1}$$

and 7 is the greatest common divisor of 126 and 35. One of the most important applications is for exact arithmetic on rational numbers, where one has to simplify $35/126$ to $5/18$ in order to keep the numbers small.

This algorithm can also be adapted to work for polynomials. It is convenient to use the following general scenario, which captures both situations under one umbrella. The reader may always think of R as being either the integers or polynomials. The algebraic terminology is explained in Chapter 25.

DEFINITION 3.1. *An integral domain R with a function $d: R \longrightarrow \mathbb{N}\cup\{-\infty\}$ is a **Euclidean domain** if for all $a,b \in R$ with $b \neq 0$, we can divide a by b with remainder, so that*

$$\text{there exist } q,r \in R \text{ such that } a = qb+r \text{ and } d(r) < d(b). \tag{2}$$

*We say that $q = a$ quo b is the **quotient** and $r = a$ rem b the **remainder**, although q and r need not be unique. Such a d is called a **Euclidean function** on R.*

EXAMPLE 3.2. (i) $R = \mathbb{Z}$ and $d(a) = |a| \in \mathbb{N}$. Here the quotient and the remainder can be made unique by the additional requirement that $r \geq 0$.

(ii) $R = F[x]$, where F is a field, and $d(a) = \deg a$. We define the degree of the zero polynomial to be $-\infty$. It is easy to show uniqueness of the quotient and the remainder in this case (Section 2.4).

(iii) $R = \mathbb{Z}[i] = \{a + ib: a, b \in \mathbb{Z}\}$, the ring of Gaussian integers, with $i = \sqrt{-1}$, and $d(a+ib) = a^2 + b^2$ (Exercise 3.19).

(iv) R a field, and $d(a) = 1$ if $a \neq 0$ and $d(0) = 0$. ◇

The value $d(b)$ is never $-\infty$ except possibly when $b = 0$.

DEFINITION 3.3. *Let R be a ring and $a, b, c \in R$. Then c is a **greatest common divisor** (or gcd) of a and b if*

(i) $c \mid a$ and $c \mid b$,

(ii) if $d \mid a$ and $d \mid b$, then $d \mid c$, for all $d \in R$.

*Similarly, c is called a **least common multiple** (or lcm) of a and b if*

(i) $a \mid c$ and $b \mid c$,

(ii) if $a \mid d$ and $b \mid d$, then $c \mid d$, for all $d \in R$.

*A **unit** $u \in R$ is any element with a multiplicative inverse $v \in R$, so that $uv = 1$. The elements a and b are **associate** if $a = ub$ for a unit $u \in R$; we then write $a \sim b$.*

For example, 3 is a gcd of 12 and 15, and 60 is an lcm of 12 and 15 in $\mathbb{Z}$. In general, neither the gcd nor the lcm are unique, but all gcds of a and b are precisely the associates of one of them, and similarly for the lcms. The only units in $\mathbb{Z}$ are 1 and -1, and 3 and -3 are all gcds of 12 and 15 in $\mathbb{Z}$. For $R = \mathbb{Z}$, we may define $\gcd(a, b)$ as the unique nonnegative greatest common divisor and $\operatorname{lcm}(a, b)$ as the unique nonnegative least common multiple of a and b. As an example, for negative $a \in \mathbb{Z}$ we then have $\gcd(a, a) = \gcd(a, 0) = -a$. We say that two integers a, b are **coprime** (or relatively prime) if their gcd is a unit.

Greatest common divisors and least common multiples need not exist in an arbitrary ring; for an example, see Section 25.2. In the following section, however, we will prove that a gcd always exists in a Euclidean domain, and as a consequence an lcm also exists.

LEMMA 3.4. *The* gcd *in* $\mathbb{Z}$ *has the following properties, for all* $a,b,c \in \mathbb{Z}$.

(i) $\gcd(a,b) = |a| \iff a \mid b$,

(ii) $\gcd(a,a) = \gcd(a,0) = |a|$ *and* $\gcd(a,1) = 1$,

(iii) $\gcd(a,b) = \gcd(b,a)$ *(commutativity),*

(iv) $\gcd(a,\gcd(b,c)) = \gcd(\gcd(a,b),c)$ *(associativity),*

(v) $\gcd(c \cdot a, c \cdot b) = |c| \cdot \gcd(a,b)$ *(distributivity),*

(vi) $|a| = |b| \Longrightarrow \gcd(a,c) = \gcd(b,c)$.

For a proof, see Exercise 3.3. Because of the associativity, we may write

$$\gcd(a_1,\ldots,a_n) = \gcd(a_1,\gcd(a_2,\ldots,\gcd(a_{n-1},a_n)\ldots)).$$

The following algorithm computes greatest common divisors in an arbitrary Euclidean domain. The nonuniqueness of quotient and remainder complicates the description a bit. For now, "$\gcd(a,b)$" stands for any element which is a gcd of a and b, and we assume some functions quo and rem which associate to any two elements $a,b \in R$ with $b \neq 0$ unique $q = a$ quo b and $r = a$ rem b in R with $a = qb + r$ and $d(r) < d(b)$. Section 3.4 fixes the notation so that gcd has only a single value.

ALGORITHM 3.5 Traditional Euclidean Algorithm.
Input: $f, g \in R$, where R is a Euclidean domain with Euclidean function d.
Output: A greatest common divisor of f and g.

1. $r_0 \longleftarrow f, \quad r_1 \longleftarrow g$
2. $i \longleftarrow 1$
   while $r_i \neq 0$ **do** $r_{i+1} \longleftarrow r_{i-1}$ rem $r_i, \quad i \longleftarrow i+1$
3. **return** r_{i-1}.

For $f = 126$ and $g = 35$, the algorithm works precisely as illustrated at the beginning of this section.

3.2. The Extended Euclidean Algorithm

The following extension of Algorithm 3.5 computes not only the gcd but also a representation of it as a linear combination of the inputs. It generalizes the representation

$$7 = 21 - 1 \cdot 14 = 21 - (35 - 1 \cdot 21) = 2 \cdot (126 - 3 \cdot 35) - 35 = 2 \cdot 126 - 7 \cdot 35,$$

which is obtained by reading the lines of (1) from the bottom up. This important method is called the **Extended Euclidean Algorithm** and works in any Euclidean domain. In various incarnations, it plays a central role throughout this book.

ALGORITHM 3.6 Traditional Extended Euclidean Algorithm.
Input: $f, g \in R$, where R is a Euclidean domain.
Output: $\ell \in \mathbb{N}$, $r_i, s_i, t_i \in R$ for $0 \le i \le \ell+1$, and $q_i \in R$ for $1 \le i \le \ell$, as computed below.

1. $r_0 \longleftarrow f, \quad s_0 \longleftarrow 1, \quad t_0 \longleftarrow 0,$
   $r_1 \longleftarrow g, \quad s_1 \longleftarrow 0, \quad t_1 \longleftarrow 1$
2. $i \longleftarrow 1$
   while $r_i \ne 0$ **do**
   $q_i \longleftarrow r_{i-1}$ quo r_i
   $r_{i+1} \longleftarrow r_{i-1} - q_i r_i$
   $s_{i+1} \longleftarrow s_{i-1} - q_i s_i$
   $t_{i+1} \longleftarrow t_{i-1} - q_i t_i$
   $i \longleftarrow i+1$
3. $\ell \longleftarrow i-1$
   return ℓ, r_i, s_i, t_i for $0 \le i \le \ell+1$, and q_i for $1 \le i \le \ell$

We note that the algorithm terminates because the $d(r_i)$ are strictly decreasing nonnegative integers for $1 \le i \le \ell$, where d is the Euclidean function on R. The elements r_i for $0 \le i \le \ell+1$ are the **remainders** and the q_i for $1 \le i \le \ell$ are the **quotients** in the traditional (Extended) Euclidean Algorithm. The elements r_i, s_i, and t_i form the ***i*th row** in the traditional Extended Euclidean Algorithm, for $0 \le i \le \ell+1$. The central property is that $s_i f + t_i g = r_i$ for all i; in particular, $s_\ell f + t_\ell g = r_\ell$ is a gcd of f and g (see Lemma 3.8 below). We will see later that *all* other intermediate results computed by the algorithm are useful for various tasks in computer algebra.

EXAMPLE 3.7. (i) As in (1), we consider $R = \mathbb{Z}$, $f = 126$, and $g = 35$. The following table illustrates the computation.

i	q_i	r_i	s_i	t_i
0		126	1	0
1	3	35	0	1
2	1	21	1	−3
3	1	14	−1	4
4	2	7	2	−7
5		0	−5	18

We can read off row 4 that $\gcd(126, 35) = 7 = 2 \cdot 126 + (-7) \cdot 35$.

(ii) $R = \mathbb{Q}[x]$, $f = 18x^3 - 42x^2 + 30x - 6$, $g = -12x^2 + 10x - 2$. Then the computation of the traditional Extended Euclidean Algorithm goes as follows. Row $i+1$ is obtained from the two preceding ones by first computing the quotient $q_i = r_{i-1}$ quo r_i and then for each of the three remaining columns by subtracting the quotient times the entry in row i of that column from the entry in row $i-1$.

i	q_i	r_i	s_i	t_i
0		$18x^3 - 42x^2 + 30x - 6$	1	0
1	$-\frac{3}{2}x + \frac{9}{4}$	$-12x^2 + 10x - 2$	0	1
2	$-\frac{8}{3}x + \frac{4}{3}$	$\frac{9}{2}x - \frac{3}{2}$	1	$\frac{3}{2}x - \frac{9}{4}$
3		0	$\frac{8}{3}x - \frac{4}{3}$	$4x^2 - 8x + 4$

We have $\ell = 2$, and from row 2, we find that a gcd of f and g is

$$\frac{9}{2}x - \frac{3}{2} = 1 \cdot (18x^3 - 42x^2 + 30x - 6) + \left(\frac{3}{2}x - \frac{9}{4}\right)(-12x^2 + 10x - 2). \diamond$$

For a global view of the algorithm, it is convenient to consider the matrices

$$R_0 = \begin{pmatrix} s_0 & t_0 \\ s_1 & t_1 \end{pmatrix}, \quad Q_i = \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} \text{ for } 1 \le i \le \ell$$

in $R^{2\times 2}$, and $R_i = Q_i \cdots Q_1 R_0$ for $0 \le i \le \ell$. The following lemma collects some invariants of the traditional Extended Euclidean Algorithm.

LEMMA 3.8. *For $0 \le i \le \ell$, we have*

(i) $R_i \cdot \begin{pmatrix} f \\ g \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}$,

(ii) $R_i = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix}$,

(iii) $\gcd(f, g) \sim \gcd(r_i, r_{i+1}) \sim r_\ell$,

(iv) $s_i f + t_i g = r_i$ *(this also holds for $i = \ell + 1$),*

(v) $s_i t_{i+1} - t_i s_{i+1} = (-1)^i$,

(vi) $\gcd(r_i, t_i) \sim \gcd(f, t_i)$,

(vii) $f = (-1)^i (t_{i+1} r_i - t_i r_{i+1})$, $g = (-1)^{i+1} (s_{i+1} r_i - s_i r_{i+1})$,

with the convention that $r_{\ell+1} = 0$.

PROOF. For (i) and (ii) we proceed by induction on i. The case $i = 0$ is clear from step 1 of the algorithm, and we may assume $i \geq 1$. Then

$$Q_i \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i-1} - q_i r_i \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix},$$

and (i) follows from $R_i = Q_i R_{i-1}$ and the induction hypothesis. Similarly, (ii) follows from

$$Q_i \begin{pmatrix} s_{i-1} & t_{i-1} \\ s_i & t_i \end{pmatrix} = \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix}$$

and the induction hypothesis.

For (iii), let $i \in \{0, \ldots, \ell\}$. We conclude from (i) that

$$\begin{pmatrix} r_\ell \\ 0 \end{pmatrix} = Q_\ell \cdots Q_{i+1} R_i \begin{pmatrix} f \\ g \end{pmatrix} = Q_\ell \cdots Q_{i+1} \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}.$$

Comparing the first entry on both sides, we see that r_ℓ is a linear combination of r_i and r_{i+1}, and hence any common divisor of r_i and r_{i+1} divides r_ℓ. On the other hand, $\det Q_i = -1$ and the matrix Q_i is invertible over R, with inverse

$$Q_i^{-1} = \begin{pmatrix} q_i & 1 \\ 1 & 0 \end{pmatrix},$$

and hence

$$\begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix} = Q_{i+1}^{-1} \cdots Q_\ell^{-1} \begin{pmatrix} r_\ell \\ 0 \end{pmatrix}.$$

Thus both r_i and r_{i+1} are divisible by r_ℓ, and $r_\ell \sim \gcd(r_i, r_{i+1})$. In particular, this is true for $i = 0$, so that $\gcd(f, g) \sim \gcd(r_0, r_1) \sim r_\ell$.

The claim (iv) follows immediately from (i) and (ii), and (v) follows from (ii) by taking determinants:

$$\begin{aligned} s_i t_{i+1} - t_i s_{i+1} = \det \begin{pmatrix} s_i & t_i \\ s_{i+1} & t_{i+1} \end{pmatrix} &= \det R_i \\ &= \det Q_i \cdots \det Q_1 \cdot \det \begin{pmatrix} s_0 & t_0 \\ s_1 & t_1 \end{pmatrix} = (-1)^i. \end{aligned}$$

In particular, this implies that $\gcd(s_i, t_i) \sim 1$ and that R_i is invertible. Now let $p \in R$ be a divisor of t_i. If $p \mid f$, then clearly $p \mid s_i f + t_i g = r_i$. On the other hand, if $p \mid r_i$, then p also divides $s_i f = r_i - t_i g$, and hence p divides f since s_i and t_i are coprime. This proves (vi). For (vii), we multiply both sides of (i) by R_i^{-1} and obtain

$$\begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = R_i^{-1} \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix} = (-1)^i \begin{pmatrix} t_{i+1} & -t_i \\ -s_{i+1} & s_i \end{pmatrix} \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix},$$

using (ii) and (v), and the claim follows by writing this out as a system of linear equations. □

COROLLARY 3.9.
Any two elements f,g of a Euclidean domain R have a gcd $h \in R$, *and it is expressible as a linear combination $h = sf + tg$ with $s,t \in R$.*

3.3. Cost analysis for $\mathbb{Z}$ and $F[x]$

We want to analyze the cost of the traditional Extended Euclidean Algorithm 3.6 for $f,g \in R$ with $n = d(f) \geq d(g) = m \geq 0$. The number ℓ of division steps is obviously bounded by $\ell \leq d(g)+1$. We investigate the two important cases $R = F[x]$ and $R = \mathbb{Z}$ separately, starting with $R = F[x]$, where F is a field, and $d(a) = \deg a$ as usual.

We let $n_i = \deg r_i$ for $0 \leq i \leq \ell+1$, with $r_{\ell+1} = 0$. Then $n_0 = n \geq n_1 = m > n_2 > \cdots > n_\ell$, and $\deg q_i = n_{i-1} - n_i$ for $1 \leq i \leq \ell$. According to Section 2.4, we can divide the polynomial r_{i-1} of degree n_{i-1} by the polynomial r_i of degree $n_i \leq n_{i-1}$ with remainder using at most $(2n_i+1)(n_{i-1}-n_i+1)$ additions and multiplications plus one inversion in F. (Recall that we count a subtraction as an addition.) Thus the total cost for the traditional Euclidean Algorithm, that is, for computing only the r_i and q_i, including a gcd of f and g, is

$$\sum_{1 \leq i \leq \ell} (2n_i+1)(n_{i-1}-n_i+1) \tag{3}$$

additions and multiplications plus $\ell \leq m+1$ inversions in F. We first evaluate the expression above for the **normal** case where the degree drops exactly by 1 at each step, so that $n_i = m-i+1$ for $2 \leq i \leq \ell = m+1$, and later show that this is the worst case. Since $n_{i-1}-n_i+1 = 2$ for $i \geq 2$ and $n_1 = m$, (3) simplifies to

$$\begin{aligned} &(2m+1)(n-m+1) + 2 \sum_{2 \leq i \leq m+1} (2(m-i+1)+1) \\ &= (2m+1)(n-m+1) + 2(m^2-m) + 2m = 2nm+n+m+1. \end{aligned} \tag{4}$$

We now consider the sum $\sigma(n_0,n_1,\ldots,n_\ell)$ in (3) as a function of the integers $n_0 \geq n_1 > n_2 > \cdots > n_\ell \geq 0$ and show that it increases if we insert an additional integer $n_{j-1} > k > n_j$ for some $j \in \{2,\ldots,\ell\}$ or append some integer $n_\ell > k \geq 0$:

$$\begin{aligned} &\sigma(n_0,\ldots,n_{j-1},k,n_j,\ldots,n_\ell) - \sigma(n_0,\ldots,n_\ell) \\ &\quad = (2k+1)(n_{j-1}-k+1) + (2n_j+1)(k-n_j+1) - (2n_j+1)(n_{j-1}-n_j+1) \\ &\quad = 2(n_{j-1}-k)(k-n_j) + 2k+1 > 0. \end{aligned}$$

A similar argument works for $n_\ell > k \geq 0$. Proceeding inductively, we find that $\sigma(n_0,n_1,n_2,\ldots,n_\ell) < \sigma(n_0,n_1,n_1-1,\ldots,1,0)$, and conclude that the bound (4) is valid in any case.

It remains to determine the cost for computing s_i,t_i on the way.

LEMMA 3.10.

$$\deg s_i = \sum_{2 \le j < i} \deg q_j = n_1 - n_{i-1} \text{ for } 2 \le i \le \ell + 1, \tag{5}$$

$$\deg t_i = \sum_{1 \le j < i} \deg q_j = n_0 - n_{i-1} \text{ for } 1 \le i \le \ell + 1. \tag{6}$$

PROOF. We only prove the first equality; the second can be verified in the same way (Exercise 3.21 (i)). We show (5) and

$$\deg s_{i-1} < \deg s_i \text{ for } 2 \le i \le \ell + 1 \tag{7}$$

by simultaneous induction on i. For $i = 2$, we find that $s_2 = s_0 - q_1 s_1 = 1 - q_1 \cdot 0 = 1$, and $\deg s_1 = -\infty < 0 = \deg s_2$. Now we assume that $i \ge 2$ and that the claims are already proven for $2 \le j \le i$. Then, by the induction hypothesis (7), we have

$$\deg s_{i-1} < \deg s_i < n_{i-1} - n_i + \deg s_i = \deg(q_i s_i),$$

which implies that

$$\deg s_{i+1} = \deg(s_{i-1} - q_i s_i) = \deg q_i + \deg s_i > \deg s_i,$$

and

$$\deg s_{i+1} = \deg q_i + \deg s_i = \sum_{2 \le j < i} \deg q_j + \deg q_i = \sum_{2 \le j < i+1} \deg q_j,$$

where we used the induction hypothesis (5). □

THEOREM 3.11.

The traditional Extended Euclidean Algorithm 3.6 for polynomials $f, g \in F[x]$ with $\deg f = n \ge \deg g = m$ can be performed with

- *at most $m+1$ inversions and $2nm + O(n)$ additions and multiplications in F if only the quotients q_i and the remainders r_i are needed,*
- *at most $m+1$ inversions and $6nm + O(n)$ additions and multiplications in F for computing all results.*

PROOF. The first claim has already been shown, and it remains to analyze the additional cost for computing the s_i and t_i. At each step, the computation of $t_{i+1} = t_{i-1} - q_i t_i$ requires at most $2 \deg q_i \deg t_i + \deg q_i + \deg t_i + 1$ field operations for the product (Section 2.3), plus at most $\deg t_{i+1} + 1$ operations for the subtraction. Using Lemma 3.10, we obtain

$$\sum_{2 \le i \le \ell} \Big(2(n_{i-1} - n_i)(n_0 - n_{i-1}) + 2(n_0 - n_i + 1) \Big)$$

additions and multiplications in F, plus $n-m+1$ for $i=1$. In the normal case, this becomes

$$n-m+1+\sum_{2\le i\le m+1}\Big(2(n-(m-i+2))+2(n-(m-i+1)+1)\Big)$$
$$=n-m+1+4\sum_{2\le i\le m+1}(n-m+i-1)$$
$$=n-m+1+4m(n-m)+2(m^2+m)\in 4nm-2m^2+O(n).$$

A similar argument as above shows that the normal case is the worst case, so that the bound is valid in general. Finally, Exercise 3.22 (i) shows that the cost for the s_i's is at most $2(m^2+m)$, and the claim follows. □

In Chapter 11, we will find a much faster algorithm for the gcd.

Now we sketch the cost analysis when $R=\mathbb{Z}$ and $d(a)=|a|$. We may assume that $f=r_0\ge g=r_1>r_2\cdots>r_\ell\ge 0$, so that $q_i\ge 1$ for all i, and represent all numbers in 2^{64}-ary standard representation (Section 2.1). Then the length $\lambda(a)$ of a positive integer a is $\lambda(a)=\lfloor(\log a)/64\rfloor+1$, where log is the binary logarithm. But now the bound corresponding to what we used for polynomials, namely $\ell\le d(g)+1=g+1=(2^{64})^{(\log g)/64}+1\le 2^{64\lambda(g)}$, on the number of division steps in the Euclidean Algorithm for the pair $(f,g)\in\mathbb{N}^2$ is exponential in the input size $\lambda(f)+\lambda(g)$ (if $\lambda(f)$ is not much bigger than $\lambda(g)$) and hence rather useless. We can in fact prove a polynomial upper bound on ℓ, as follows. For $1\le i\le\ell$, we have

$$r_{i-1}=q_ir_i+r_{i+1}\ge r_i+r_{i+1}>2r_{i+1}.$$

Thus

$$\prod_{2\le i<\ell}r_{i-1}>2^{\ell-2}\prod_{2\le i<\ell}r_{i+1}$$

if $\ell>2$, and $r_{\ell-1}\ge 2$ implies that

$$2^{\ell-2}<\frac{r_1r_2}{r_{\ell-1}r_\ell}<\frac{r_1^2}{2},$$

$$\ell\le\lfloor 2\log r_1\rfloor+1=\left\lfloor 128\frac{\log g}{64}\right\rfloor+1\le 128\left(\left\lfloor\frac{\log g}{64}\right\rfloor+1\right)=128\lambda(g).$$

This bound can still be improved. For $N\in\mathbb{N}$ and $f,g\in\mathbb{Z}$ with $N\ge f>g>0$, the largest possible number of division steps ℓ for (f,g) is the one where all the quotients are equal to 1, so that f and g are the two largest successive Fibonacci numbers up to N. As an example, the Euclidean Algorithm for $(f,g)=(13,8)$ computes

$$\begin{aligned}13 &= 1\cdot 8+5,\\ 8 &= 1\cdot 5+3,\\ 5 &= 1\cdot 3+2,\\ 3 &= 1\cdot 2+1,\\ 2 &= 2\cdot 1.\end{aligned}$$

The nth **Fibonacci number** F_n (with $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$) is approximately $\phi^n/\sqrt{5}$, where $\phi = (1+\sqrt{5})/2 \approx 1.618$ is the **golden ratio** (Exercise 3.28). Thus the following holds for the number ℓ of division steps for $(f,g) = (F_{n+1}, F_n)$ if $n \geq 1$:

$$\ell = n-1 \approx \log_\phi \sqrt{5}g - 1 \in 1.441 \log g + O(1). \tag{8}$$

The average number of division steps for (f,g) when g is fixed and f varies is

$$\ell \approx \frac{12(\ln 2)^2}{\pi^2} \log g \approx 0.584 \log g.$$

Now that we have a good upper bound for the number of steps in the Euclidean Algorithm, we look at the cost for each step. First we consider the cost for one division step. Let $a > b > 0$ be integers and $a = qb + r$ with $q, r \in \mathbb{N}$ and $0 \leq r < b$. According to Section 2.4, computing q and r takes $O((\lambda(a) - \lambda(b)) \cdot \lambda(b))$ word operations, where $\lambda(a)$ and $\lambda(b)$ are the lengths of a and b in the standard representation, respectively.

Then setting $n = \lambda(f)$ and $m = \lambda(g)$, we obtain—by analogy with (4)—that the total cost for performing the traditional Euclidean Algorithm (without computing the s_i and t_i) is $O(nm)$ word operations.

The following integer analog of Lemma 3.10 is proven in Exercise 3.23.

LEMMA 3.12. *$|s_i| \leq \dfrac{g}{r_{i-1}}$ and $|t_i| \leq \dfrac{f}{r_{i-1}}$ for $1 \leq i \leq \ell+1$.*

Lemma 3.12 yields analogous bounds for the length of s_i and t_i as in the polynomial case, and we have the following theorem, whose proof is left as Exercise 3.24.

THEOREM 3.13.
The traditional Extended Euclidean Algorithm 3.6 for positive integers f, g with $\lambda(f) = n \geq \lambda(g) = m$ can be performed with $O(nm)$ word operations.

N	c_N/N^2
10	0.63
100	0.6087
1000	0.608383
10000	0.60794971
100000	0.6079301507

TABLE 3.1: The probabilities that two random positive integers below N are coprime.

We conclude this section with the following question: what is the probability that two random integers are coprime? More precisely, when N gets large and $c_N = \#\{1 \le x, y \le N\colon \gcd(x,y) = 1\}$, we are interested in the numerical value of c_N/N^2. Table 3.1 gives c_N/N^2 for some values of N; it seems to approach a limit which is a little larger than $3/5$. In fact, the value is

$$\frac{c_N}{N^2} \in \frac{6}{\pi^2} + O\Big(\frac{\log N}{N}\Big) \approx 0.6079271016 + O\Big(\frac{\log N}{N}\Big).$$

Interestingly, a similar approximation holds for the probability that a random integer is **squarefree**, so that it has no square divisor p^2:

$$\frac{\#\{1 \le x \le N\colon x \text{ is squarefree}\}}{N} \in \frac{6}{\pi^2} + O\Big(\frac{1}{\sqrt{N}}\Big).$$

Exercises 4.18 and 14.32 answer the corresponding questions for polynomials over a finite field.

In Figure 3.2, we see a two-dimensional coordinate system where the point $(x,y) \in \mathbb{N}^2$ for $x, y \le 200$ is colored white if $\gcd(x,y) = 1$ and gray otherwise. The intensity of a pixel is proportional to the number of prime factors in the gcd. The probability that two random integers below 200 are coprime is precisely the percentage of the area of the 200×200 pixels that is colored white. Thus about $3/5$ of all pixels are white, and about $2/5$ are gray.

If you hold the page horizontally in front of your eyes, you can see (almost) white horizontal and vertical lines corresponding to prime values of x and y, and dark lines through the origin corresponding to lines $ax = by$ with small integers a, b, the most clearly visible being the line $x = y$.

3.4. (Non-)Uniqueness of the gcd

The nonuniqueness of the gcd is a harmless nuisance from a mathematical point of view. But in software, we have to implement a *function* gcd with a unique output. In this section, we discuss one way of achieving this.

Since $\mathbb{Q}$ is a field, every nonzero rational number is a unit in $\mathbb{Q}$, and so $ua \sim a$ in $R = \mathbb{Q}[x]$ for all nonzero $u \in \mathbb{Q}$ and all $a \in R$. If we want to define a single

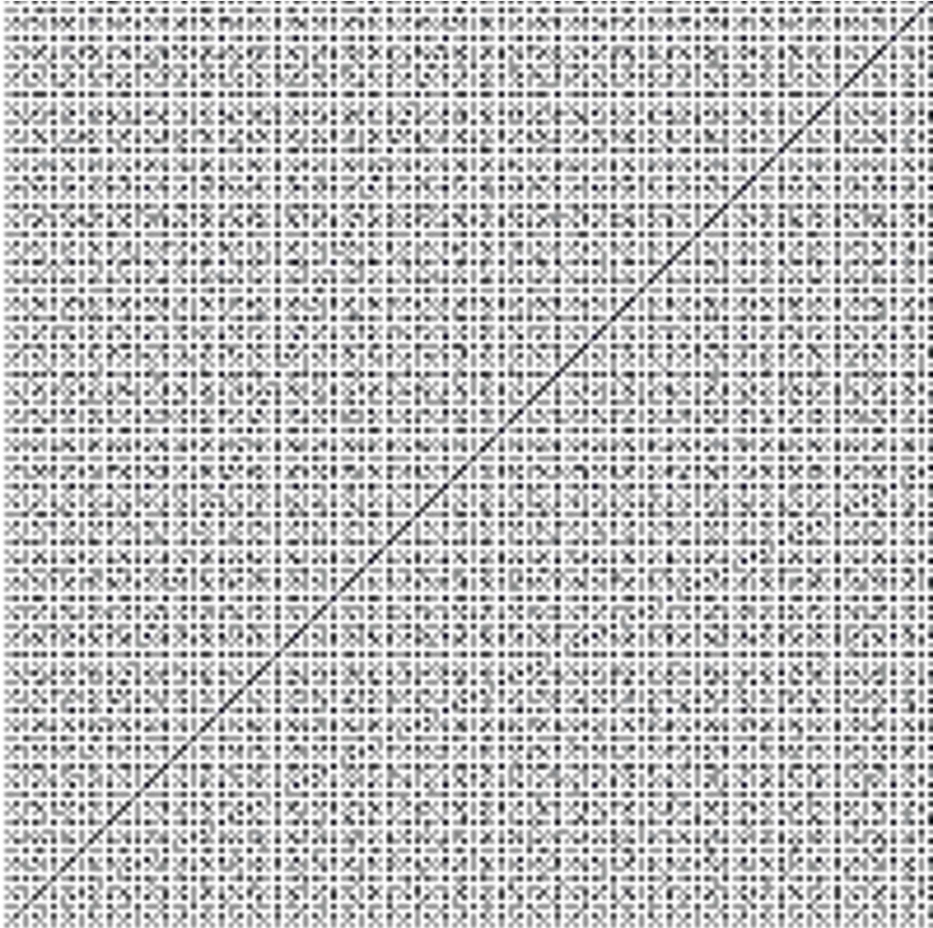


FIGURE 3.2: The greatest common divisors of x and y for $1 \leq x, y \leq 200$.

element $\gcd(f,g) \in \mathbb{Q}[x]$, which one should we choose? In other words, how do we choose *one* representative from among all the multiples of a? A reasonable choice is the **monic** polynomial, that is, the one with leading coefficient 1. Thus if $\mathrm{lc}(a) \in \mathbb{Q} \setminus \{0\}$ is the leading coefficient of $a \in \mathbb{Q}[x]$, then we take $\mathrm{normal}(a) = a/\mathrm{lc}(a)$ as the **normal form** of a. (This has nothing to do with the "normal EEA" on page 51.)

To make this work in an arbitrary Euclidean domain R, we assume that we have selected some normal form $\mathrm{normal}(a) \in R$ for every $a \in R$ so that $a \sim \mathrm{normal}(a)$. We call the unit $u \in R$ with $a = u \cdot \mathrm{normal}(a)$ the **leading unit** $\mathrm{lu}(a)$ of a. Moreover, we set $\mathrm{lu}(0) = 1$ and $\mathrm{normal}(0) = 0$. The following two properties are required:

- two elements of R have the same normal form if and only if they are associate,
- the normal form of a product is equal to the product of the normal forms.

These properties in particular imply that the normal form of any unit is 1. We say that an element a in normal form, so that $\mathrm{lu}(a) = 1$, is **normalized**.

In our two main applications, integers and univariate polynomials over a field, we have natural normal forms. If $R = \mathbb{Z}$, $\mathrm{lu}(a) = \mathrm{sign}(a)$ if $a \neq 0$ and $\mathrm{normal}(a) = |a|$ defines a normal form, so that an integer is normalized if and only if it is nonnegative. When $R = F[x]$ for a field F, then letting $\mathrm{lu}(a) = \mathrm{lc}(a)$ (with the convention that $\mathrm{lu}(0) = 1$) and $\mathrm{normal}(a) = a/\mathrm{lc}(a)$ defines a normal form, and a nonzero polynomial is normalized if and only if it is monic.

Given such a normal form, we define $\gcd(a,b)$ to be the unique normalized associate of all greatest common divisors of a and b, and similarly $\mathrm{lcm}(a,b)$ as the normalized associate of all least common multiples of a and b. Thus $\gcd(a,b) > 0$ for $R = \mathbb{Z}$ and $\gcd(a,b)$ is monic for $R = F[x]$ if at least one of a, b is nonzero, and $\gcd(0,0) = 0$ in both cases. Lemma 3.4 then remains valid if we replace $|\cdot|$ by $\mathrm{normal}(\cdot)$.

In the polynomial case, it turns out that it is not only useful to have a normal form for the gcd, but to modify the traditional Euclidean Algorithm so that *all* the remainders r_i are normalized. In Chapter 6, we will see that for $R = \mathbb{Q}[x]$ the computations of the traditional Euclidean Algorithm produce remainders whose coefficients have huge numerators and denominators even for inputs of moderate size, and that the coefficients of the monic associates of the remainders are much smaller (see pages 143 and 185). In this book, we will often use the following variant of the traditional Extended Euclidean Algorithm 3.6 which works with these monic associates.

ALGORITHM 3.14 Extended Euclidean Algorithm (EEA).

Input: $f, g \in R$, where R is a Euclidean domain with a normal form.

Output: $\ell \in \mathbb{N}$, $\rho_i, r_i, s_i, t_i \in R$ for $0 \leq i \leq \ell+1$, and $q_i \in R$ for $1 \leq i \leq \ell$, as computed below.

1. $\rho_0 \longleftarrow \mathrm{lu}(f)$, $r_0 \longleftarrow \mathrm{normal}(f)$, $s_0 \longleftarrow \rho_0^{-1}$, $t_0 \longleftarrow 0$,
   $\rho_1 \longleftarrow \mathrm{lu}(g)$, $r_1 \longleftarrow \mathrm{normal}(g)$, $s_1 \longleftarrow 0$, $t_1 \longleftarrow \rho_1^{-1}$

2. $i \longleftarrow 1$
   while $r_i \neq 0$ **do**
   $q_i \longleftarrow r_{i-1} \text{ quo } r_i$
   $\rho_{i+1} \longleftarrow \mathrm{lu}(r_{i-1} - q_i r_i)$
   $r_{i+1} \longleftarrow (r_{i-1} - q_i r_i)/\rho_{i+1}$
   $s_{i+1} \longleftarrow (s_{i-1} - q_i s_i)/\rho_{i+1}$
   $t_{i+1} \longleftarrow (t_{i-1} - q_i t_i)/\rho_{i+1}$
   $i \longleftarrow i+1$

3. $\ell \longleftarrow i-1$
   return ℓ, ρ_i, r_i, s_i, t_i for $0 \leq i \leq \ell+1$, and q_i for $1 \leq i \leq \ell$

The elements r_i for $0 \le i \le \ell+1$ are the **remainders**, the q_i for $1 \le i \le \ell$ are the **quotients**, and the elements r_i, s_i, and t_i form the **ith row** in the Extended Euclidean Algorithm, for $0 \le i \le \ell+1$. The elements s_ℓ and t_ℓ, satisfying $s_\ell f + t_\ell g = \gcd(f,g)$, are the **Bézout coefficients** of f and g.

EXAMPLE 3.7 (continued). (ii) With monic remainders, the following quantities are computed.

i	q_i	ρ_i	r_i	s_i	t_i
0		18	$x^3 - \frac{7}{3}x^2 + \frac{5}{3}x - \frac{1}{3}$	$\frac{1}{18}$	0
1	$x - \frac{3}{2}$	-12	$x^2 - \frac{5}{6}x + \frac{1}{6}$	0	$-\frac{1}{12}$
2	$x - \frac{1}{2}$	$\frac{1}{4}$	$x - \frac{1}{3}$	$\frac{2}{9}$	$\frac{1}{3}x - \frac{1}{2}$
3		1	0	$-\frac{2}{9}x + \frac{1}{9}$	$-\frac{1}{3}x^2 + \frac{2}{3}x - \frac{1}{3}$

From row 2, we find that

$$\gcd(f,g) = x - \frac{1}{3} = \frac{2}{9} \cdot (18x^3 - 42x^2 + 30x - 6) + \Big(\frac{1}{3}x - \frac{1}{2}\Big)(-12x^2 + 10x - 2). \ \diamond$$

The matrices Q_i now become:

$$Q_i = \begin{pmatrix} 0 & 1 \\ \rho_{i+1}^{-1} & -q_i\rho_{i+1}^{-1} \end{pmatrix} \text{ for } 1 \le i \le \ell.$$

LEMMA 3.15. *(a) With the following modifications, all statements of Lemma 3.8 hold for the results of Algorithm 3.14.*

(iii) $\gcd(f,g) = \gcd(r_i, r_{i+1}) = r_\ell$,

(v) $s_i t_{i+1} - t_i s_{i+1} = (-1)^i (\rho_0 \cdots \rho_{i+1})^{-1}$,

(vi) $\gcd(r_i, t_i) = \gcd(f, t_i)$,

(vii) $f = (-1)^i \rho_0 \cdots \rho_{i+1}(t_{i+1}r_i - t_i r_{i+1})$, $g = (-1)^{i+1}\rho_0 \cdots \rho_{i+1}(s_{i+1}r_i - s_i r_{i+1})$.

(b) If $R = F[x]$ for a field F, $\deg f \ge \deg g$, and $n_i = \deg r_i$ for all i, then the degree formulas of Lemma 3.10 hold for the results of Algorithm 3.14 as well.

PROOF. With the following changes, the proof of Lemma 3.8 goes through:

$$Q_i \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ \rho_{i+1}^{-1} & -q_i\rho_{i+1}^{-1} \end{pmatrix} \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = \begin{pmatrix} r_i \\ (r_{i-1} - q_i r_i)\rho_{i+1}^{-1} \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix},$$

$$Q_i^{-1} = \begin{pmatrix} q_i & \rho_{i+1} \\ 1 & 0 \end{pmatrix},$$

$$\det Q_i \cdots \det Q_1 \cdot \det \begin{pmatrix} s_0 & t_0 \\ s_1 & t_1 \end{pmatrix} = (-1)^i (\rho_0 \cdots \rho_{i+1})^{-1},$$

$$\begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = R_i^{-1} \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix} = (-1)^i (\rho_0 \cdots \rho_{i+1}) \begin{pmatrix} t_{i+1} & -t_i \\ -s_{i+1} & s_i \end{pmatrix} \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}.$$

Statements (iii) and (vi) follow from the fact that all elements involved are normalized. The proof of (b) is left as Exercise 3.21 (ii). □

We conclude this section with a cost analysis of the EEA for polynomials. It turns out to be not more expensive than the traditional EEA.

THEOREM 3.16.
For the monic normal form $\mathrm{normal}(h) = h/\mathrm{lc}(h)$ *on* $F[x]$, *the Extended Euclidean Algorithm 3.14 for polynomials* $f, g \in F[x]$ *with* $\deg f = n \geq \deg g = m$ *can be performed with*

- *at most* $m+2$ *inversions and* $2nm + O(n)$ *additions and multiplications in* F *if only the quotients* q_i, *the remainders* r_i, *and the coefficients* ρ_i *are needed,*
- *at most* $m+2$ *inversions and* $6nm + O(n)$ *additions and multiplications in* F *for computing all results.*

PROOF. We proceed as in Section 3.3 and let $n_i = \deg r_i$ for $0 \leq i \leq \ell + 1$, with $r_{\ell+1} = 0$, so that $\deg q_i = n_{i-1} - n_i$ for $1 \leq i \leq \ell$. Division with remainder of a monic polynomial a by a monic polynomial b is slightly cheaper than a general division; it takes only $2 \deg b \cdot (\deg a - \deg b) + \deg b$ additions and multiplications in F. Thus the cost for computing the quotient q_i and the remainder on division of r_{i-1} by r_i is $2n_i(n_{i-1} - n_i) + n_i$. If $i < \ell$, we compute ρ_{i+1}^{-1} and multiply the remainder by it to obtain r_{i+1}, taking one inversion plus n_{i+1} multiplications. Thus the cost for computing all q_i and r_{i+1} for $1 \leq i \leq \ell$ is $\ell - 1 \leq m$ inversions and

$$\sum_{1 \leq i \leq \ell} \Big(2n_i(n_{i-1} - n_i) + n_i \Big) + \sum_{1 \leq i < \ell} n_{i+1}$$

additions and multiplications. For a normal degree sequence, where $n_i = m - i + 1$ for $1 \leq i \leq \ell = m + 1$, this becomes

$$2m(n-m) + m + \sum_{2 \leq i \leq m+1} 4(m - i + 1) = 2m(n-m) + m + 2(m^2 - m) = 2nm - m.$$

As in Section 3.3, the normal case is the worst case, normalizing f and g takes two inversions and $n + m$ multiplications, and (i) follows.

As for division with remainder, multiplying a monic polynomial a by a polynomial b is a bit cheaper than a general multiplication; it takes only $2\deg a \cdot \deg b + \deg a$ additions and multiplications in F (Section 2.3). Thus computing $q_i t_i$ takes $2(n_{i-1}-n_i)(n-n_{i-1})+n_{i-1}-n_i$ operations, by Lemma 3.15 (b). Subtracting the product from t_{i-1} and multiplying the result by ρ_{i+1}^{-1} to obtain t_{i+1} takes another $2(n-n_i+1)$ additions and multiplications. Hence the cost for computing all t_i is

$$\sum_{1\leq i\leq \ell} \Big(2(n_{i-1}-n_i)(n-n_{i-1})+n_{i-1}-n_i+2(n-n_i+1)\Big)$$

additions and multiplications in F. In the normal case, this simplifies to

$$\begin{aligned}\sum_{1\leq i\leq m+1} &\Big(2(n-(m-i+2))+1+2(n-(m-i+1)+1)\Big)\\ &= m+1+\sum_{1\leq i\leq m+1} 4(n-m+i-1)\\ &= m+1+4(m+1)(n-m)+2(m^2+m) = 4nm-2m^2+4n-m+1.\end{aligned}$$

Exercise 3.22 (ii) shows that the cost for computing all s_i is at most $2m^2+3m+1$ in the normal case. Again, the normal case is the worst case, and (ii) follows. $\square$

Theorem 6.53 (i) in Section 6.11 shows that in the polynomial case, the results of the traditional EEA and the results of Algorithm 3.14 are constant multiples of each other.

Taking the positive or monic gcd in $\mathbb{Z}$ or $F[x]$, respectively, is a reasonable solution to the nonuniqueness problem. However, when you implement computer algebra software, many other rings will be relevant, and often normalization is not compatible across domains. For example, $\gcd(-10x, 5x^2)$ is not really defined unless we specify the domain R in Definition 3.3. Using R as a subscript, we have—under normalization—$\gcd_{\mathbb{Q}[x]}(-10x, 5x^2) = x$, and $\pm 5x$ are candidates for $\gcd_{\mathbb{Z}[x]}(-10x, 5x^2)$. A computer algebra system has to make an assumption here, unless it allows the user to specify the domain; for our example, usually $\mathbb{Z}[x]$ is assumed.

If R is a domain with a normal form normal_R, then we get one for the polynomial ring $R[x]$ by setting

$$\mathrm{normal}_{R[x]}(f) = \frac{\mathrm{normal}_R(\mathrm{lc}(f))}{\mathrm{lc}(f)} \cdot f,$$

where $\mathrm{lc}(f)$ is the leading coefficient of f (Exercise 3.8 (iii)). Inductively, this defines a normal form, and hence a unique gcd, for multivariate polynomials over $\mathbb{Z}$ or over any field.

Notes. 3.1. The algorithm described in Euclid's *Elements* does not use division with remainder, but rather subtracts the smaller number g from the larger one until it becomes smaller than g, and then swaps the two.

Allowing $-\infty$ as a value of a Euclidean function d is a bit annoying and makes our two main examples, integers and univariate polynomials over a field, look different. The proper analogy between $\mathbb{Z}$ and $F[x]$ goes as follows. We can take $d(a) = |a|$ on $\mathbb{Z}$ and $d(a) = 2^{\deg a}$ on $F[x]$, including $d(0) = 0$ in both cases; then $d(ab) = d(a)d(b)$. Or, equivalently, we can take $d(a) = \lfloor \log_2 |a| \rfloor$ on $\mathbb{Z}$ (Exercise 3.5) and $d(a) = \deg a$ on $F[x]$, with $d(0) = -\infty$ in both cases; then $d(ab)$ is $d(a) + d(b)$ (or $d(a) + d(b) + 1$ in $\mathbb{Z}$).

3.2. The astronomical book Āryabhaṭīya, written by Āryabhaṭa in Sanskrit near the end of the fifth century AD, contains an algorithm for computing from two coprime integers $f, g \in \mathbb{N}$ two integers s, t such that $sf + tg = 1$. This problem is also solved in Bachet (1612).

Exercise 3.25 discusses the binary Euclidean Algorithm of Stein (1967). Knuth (1998), already in the second edition, states a binary EEA due to Michael Penk (Algorithm Y in the Answers to Exercises of §4.5.2). Weilert (2000) adapts the binary Euclidean Algorithm to the Gaussian integers.

Although the polynomial version of the (Extended) Euclidean Algorithm is conceptually somewhat simpler, it is much younger (Stevin 1585; Newton 1707, page 38) than the 2000-year old integer algorithm. One reason for this is that we have a more intuitive understanding of integers than we do of polynomials.

3.3. The fact that the number of division steps is maximal for Fibonacci numbers is Lamé's (1844) theorem. The scholarly work of Bach & Shallit (1996) contains more complete historical information about this and many other topics in this book. The interesting paper by Shallit (1994) points to three earlier analyses of the number of divisions in Euclid's algorithm: Reynaud (1824), Finck (1841), and Binet (1841); the latter allows negative remainders, as in Exercise 3.13. Finck's wording *un problème qui* [...] *a pour objet de déterminer le nombre des opérations de la recherche du p.g.c.d. de deux nombres entiers*[1] is a remarkably modern-sounding demand for the analysis of Euclid's algorithm. He gives the inequality $r_{i-1} > 2r_{i+1}$ that we used. Dupré (1846) gives the bounds of about $(\log f)/\log((1+\sqrt{5})/2)$ for the ordinary and $(\log f)/\log(1+\sqrt{2})$ for Binet's Euclidean Algorithm (Exercise 3.30). Much earlier, Schwenter (1636), 86. Auffgab, calls the Euclidean Algorithm for 770 020 512 197 390 and 124 591 930 070 091, with 32 divisions, the *arithmetical labyrinth*, due to Simon Jacob von Coburg, and points to the Fibonacci numbers as requiring many divisions in the Euclidean Algorithm. (The two large integers are not Fibonacci numbers, and Schwenter says that their Euclidean Algorithm requires 54 divisions; there is a calculation or copying mistake somewhere.) We have $\gcd(F_n, F_m) = F_{\gcd(n,m)}$; see Exercise 3.31.

The average number of division steps in the Euclidean Algorithm for integers was investigated by Heilbronn (1968) and Dixon (1970), and in the binary algorithm (Exercise 3.25) by Brent (1976); see Knuth (1998), §4.5.2, and Shallit (1994) for surveys. Those results were all based on reasonable but unproven assumptions. The question was finally settled by Vallée (2003); she gives average case analyses of several variations, with about $\frac{\pi^2}{6}n$ many divisions on average for the Euclidean Algorithm on n-bit numbers. For polynomials over

[1] a problem that [...] has as its goal to determine the number of operations in computing the gcd of two integers

finite fields, Ma & von zur Gathen (1990)) give worst case and average case analyses of several variants of the Euclidean Algorithm.

The fact that two random integers are coprime with probability $6/\pi^2$ is a theorem of Dirichlet (1849). Dirichlet also proves the fact, surprising at first sight, that for fixed a in a division the remainder $r = a \operatorname{rem} b$, with $0 \leq r < b$, is more likely to be smaller than $b/2$ than larger: If p_a denotes the probability for the former, where $1 \leq b \leq a$ is chosen uniformly at random, then p_a is asymptotically $2 - \ln 4 \approx 61.37\%$. For Dirichlet's theorem, and also the corresponding statement about the probability of being squarefree (due to Gegenbauer 1884), see Hardy & Wright (1985), §§18.5 and 18.6. A heuristic argument goes as follows. A prime p divides a random integer x with probability $1/p$, and neither x nor y with probability $1 - 1/p^2$. Hence $\gcd(x,y) = 1$ happens with probability $\zeta(2)^{-1} = \prod_{p \text{ prime}}(1 - 1/p^2) = 6/\pi^2$; see Notes 18.4 for a discussion of Riemann's zeta function. The value of $\zeta(2)$ was determined by Euler (1734/35b, 1743); see Apostol (1983) for a simple way of calculating this quantity.

3.4. The Euclidean Algorithm 3.14 with monic remainders (for univariate polynomials) appears in the 1969 edition of Knuth (1998), and in Brown (1971).

The calculation of the Bézout coefficients via the EEA in general is in Euler (1748a), §70. See also Notes 6.3. Gauß (1863b), articles 334 and 335, does this for polynomials in $\mathbb{F}_p[x]$, where p is prime.

Exercises.

3.1 Prove that two odd integers whose difference is 32 are coprime.

3.2 Let R be an integral domain. Show that

$$a \sim b \iff (a \mid b \text{ and } b \mid a) \iff \langle a \rangle = \langle b \rangle,$$

where $\langle a \rangle = Ra = \{ra : r \in R\}$ is the ideal generated by a.

3.3 Prove Lemma 3.4. Hint: For (v) and (vi), show that any divisor of the left hand side also divides the right hand side, and vice versa. What are the corresponding statements for the lcm? Are they also true?

3.4* Show that $\gcd(a,b) = 1$ and $\gcd(a,c) = 1$ imply that $\gcd(a,bc) = 1$.

3.5** We consider the following property of a Euclidean function on an integral domain R:

$$d(ab) \geq d(b) \text{ for all } a,b \in R \setminus \{0\}. \tag{9}$$

Our two familiar examples, the degree on $F[x]$ for a field F and the absolute value on $\mathbb{Z}$, both fulfill this property. This exercise shows that every Euclidean domain has such a Euclidean function.

(i) Show that $\delta : \mathbb{Z} \longrightarrow \mathbb{N}$ with $\delta(3) = 2$ and $\delta(a) = |a|$ if $a \neq 3$ is a Euclidean function on $\mathbb{Z}$ violating (9).

(ii) Suppose that R is a Euclidean domain and $D = \{\delta : \delta \text{ is a Euclidean function on } R\}$. Then D is nonempty, and we may define a function $d : R \longrightarrow \mathbb{N} \cup \{-\infty\}$ by $d(a) = \min\{\delta(a) : \delta \in D\}$. Show that d is a Euclidean function on R (called the **minimal Euclidean function**).

(iii) Let δ be a Euclidean function on R such that $\delta(ab) < \delta(b)$ for some $a,b \in R \setminus \{0\}$. Find another Euclidean function δ^* that is smaller than δ. Conclude that the minimal Euclidean function d satisfies (9).

(iv) Show that for all $a,b \in R \setminus \{0\}$ and a Euclidean function d satisfying (9), we have $d(0) < d(a)$, and $d(ab) = d(b)$ if and only if a is a unit.

(v) Let d be the minimal Euclidean function as in (ii). Conclude that $d(0) = -\infty$ and the group of units of R is $R^\times = \{a \in R \setminus \{0\} : d(a) = 0\}$.

(vi) Prove that $d(a) = \deg a$ is the minimal Euclidean function on $F[x]$ for a field F, and that $d(a) = \lfloor \log_2 |a| \rfloor$ is the minimal Euclidean function on $\mathbb{Z}$, with $d(0) = -\infty$ in both cases.

3.6* (i) Show that each two nonzero elements a, b of a UFD R have a gcd as well as a lcm. You may assume that a normal form on R is given (this is not a restriction, by Exercise 3.9). Hint: First look at the special case $R = \mathbb{Z}$, and use the factorizations of $\mathrm{normal}(a)$ and $\mathrm{normal}(b)$ into normalized primes.

(ii) Prove that $\gcd(a,b) \cdot \mathrm{lcm}(a,b) = \mathrm{normal}(a \cdot b)$.

(iii) Conclude that $\mathrm{lcm}(a_1, \ldots, a_n) = \mathrm{normal}(a_1 \cdots a_n)$ for any n nonzero elements $a_1, \ldots, a_n \in R$ that are pairwise coprime (you might need Exercise 3.4).

(iv) Is $\gcd(a_1, \ldots, a_n) \cdot \mathrm{lcm}(a_1, \ldots, a_n) = \mathrm{normal}(a_1 \cdots a_n)$ valid for arbitrary $n \in \mathbb{N}$?

3.7* Let R be a Euclidean domain, with a Euclidean function $d : R \longrightarrow \mathbb{N} \cup \{-\infty\}$ that has the additional properties

- $d(ab) = d(a) + d(b)$,
- $d(a+b) \le \max\{d(a), d(b)\}$, with equality if $d(a) \ne d(b)$,
- d is surjective,

for all $a, b \in R$. Prove that R is a polynomial ring with d as degree function. Proceed as follows:

(i) Prove that $d(a) = -\infty$ if and only if $a = 0$.

(ii) Show that $F = \{a \in R : d(a) \le 0\}$ is a subfield of R.

(iii) Let $x \in R$ be such that $d(x) = 1$, and prove that every nonzero $a \in R$ has a unique representation

$$a = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0,$$

where $n = d(a)$, $a_0, \ldots, a_n \in F$, and $a_n \ne 0$.
Hint: You may find Exercise 3.5 (iv) useful.

3.8 (i) Find normal forms for $\mathbb{Z}[x]$ and for a field F.

(ii) Now let R be an integral domain with a normal form. Prove that $\mathrm{lu}(ua) = u \cdot \mathrm{lu}(a)$ holds for all units $u \in R$ and all $a \in R$. Show that a is normalized if and only if $\mathrm{normal}(a) = a$.

(iii) Prove that $\mathrm{lu}_{R[x]}(f) = \mathrm{lu}_R(f_n)$ for a polynomial $f = \sum_{0 \le i \le n} f_i x^i \in R[x]$ with $f_n \ne 0$ defines a normal form on $R[x]$ which extends the given normal form on R. Describe this normal form when R is a field.

(iv) Let $a = bc \in R$ with normalized a, b. Prove that c is normalized if $b \ne 0$, and conclude that $\mathrm{normal}(a/b) = \mathrm{normal}(a)/\mathrm{normal}(b)$ holds for all $a, b \in R$ such that $a \mid b$ and $b \ne 0$.

(v) Starting with the usual normal form $\mathrm{normal}_{\mathbb{Z}}(a) = |a|$ on $\mathbb{Z}$, we can use part (iii) of this exercise in two ways to obtain a normal form on $R = \mathbb{Z}[x,y]$, namely by regarding R either as $\mathbb{Z}[x][y]$ or as $\mathbb{Z}[y][x]$. Determine both normal forms for the polynomial $x - y \in R$.

3.9* The goal of this exercise is to show that every UFD has a normal form.

(i) For a prime $p \in \mathbb{N}$, define $\mathrm{lu}(p) = 1$ if $p \ne 2$ and $\mathrm{lu}(2) = -1$. Show that "lu" can be uniquely extended to a normal form on $\mathbb{Z}$.

(ii) Now let R be an arbitrary UFD and $P \subseteq R$ a complete set of representatives for all non-associate primes of R, so that every prime of R is associate to a unique $p \in P$. (The existence of such a set is guaranteed by the axiom of choice in general.) Then every $r \in R$ can be *uniquely* written as $r = u \prod_{p \in S} p$ for a finite subset $S \subseteq R$ and a unit $u \in R$. Show that $\mathrm{lu}(r) = u$ defines a normal form on R.

3.10 Are there $s, t \in \mathbb{Z}$ such that $24s + 14t = 1$?

3.11 For each of the following pairs of integers, find their greatest common divisor using the Euclidean Algorithm:

(i) $34, 21$; (ii) $136, 51$; (iii) $481, 325$; (iv) $8771, 3206$.

3.12 Show that $\{sf + tg : s,t \in \mathbb{Z}\} = \{k \cdot \gcd(f,g) : k \in \mathbb{Z}\}$ holds for all $f, g \in \mathbb{Z}$. (In other words, the two ideals $\langle f, g\rangle$ and $\langle \gcd(f,g)\rangle$ are identical.)

3.13 The Euclidean Algorithm for integers can be slightly speeded up if it is permitted to carry out divisions with negative remainders, so that $r_{i-1} = r_i q_i + r_{i+1}$ with $-|r_i/2| < r_{i+1} \le |r_i/2|$. Do the four examples in Exercise 3.11 using this method.

3.14 Use the Extended Euclidean Algorithm to find $\gcd(f,g)$, for $f, g \in \mathbb{Z}_p[x]$ in each of the following examples (arithmetic in $\mathbb{Z}_p = \{0, \ldots, p-1\}$ is done modulo p). In each case compute the corresponding polynomials s and t such that $\gcd(f,g) = sf + tg$.

(i) $f = x^3 + x + 1, g = x^2 + x + 1$ for $p = 2$ and $p = 3$.

(ii) $f = x^4 + x^3 + x + 1$, $g = x^3 + x^2 + x + 1$ for $p = 2$ and $p = 3$.

(iii) $f = x^5 + x^4 + x^3 + x + 1$, $g = x^4 + x^3 + x^2 + x + 1$ for $p = 5$.

(iv) $f = x^5 + x^4 + x^3 - x^2 - x + 1$, $g = x^3 + x^2 + x + 1$ for $p = 3$ and $p = 5$.

3.15 Show that the s_i and t_i in the traditional Extended Euclidean Algorithm for inputs $f, g \in \mathbb{Z}$ with $f > g > 0$ alternate in sign, so that s_{2i} and t_{2i-1} are positive and s_{2i+1} and t_{2i} are negative for all admissible values of $i \ge 1$. Conclude that $0 = s_1 < 1 = s_2 \le |s_3| < |s_4| < \cdots < |s_{\ell+1}|$ and $0 = t_0 < 1 = t_1 \le |t_2| < |t_3| < \cdots < |t_{\ell+1}|$.

3.16 Let R be a Euclidean domain, $a, b, c \in R$, and $\gcd(a,b) = 1$. Prove the following:

(i) $a \mid bc \Longrightarrow a \mid c$,

(ii) $a \mid c$ and $b \mid c \Longrightarrow ab \mid c$.

Hint: You may want to use the fact that the Extended Euclidean Algorithm computes $s, t \in R$ such that $sa + tb = 1$.

3.17 Prove that $\mathbb{Z}[x]$ is not a Euclidean domain. Hint: If it were, then we could compute $s, t \in \mathbb{Z}[x]$ such that $s \cdot 2 + t \cdot x = \gcd(2, x)$, using the Extended Euclidean Algorithm.

3.18* Let $R = F[x]$ for a field F and

$$S = \bigcup_{\ell \ge 1} \Big((F \setminus \{0\})^{\ell+1} \times (R \setminus \{0\})^2 \times \{q \in R : \deg q > 0,\ q \text{ monic}\}^{\ell-1} \Big).$$

The **Euclidean representation** of a pair $(f,g) \in (R \setminus \{0\})^2$ with $\deg f \ge \deg g$ is defined as the list $(\rho_0, \ldots, \rho_\ell, r_\ell, q_1, \ldots, q_\ell) \in S$ formed from the results of the Euclidean Algorithm. Show that the map

$$\{(f,g) \in R^2 : \deg f \ge \deg g \text{ and } g \ne 0\} \longrightarrow S$$

which maps a pair of polynomials (f,g) to its Euclidean representation is a bijection.

3.19* (i) Show that the **norm** $N : \mathbb{Z}[i] \longrightarrow \mathbb{N}$ with $N(\alpha) = \alpha\overline{\alpha}$ on the ring of Gaussian integers $\mathbb{Z}[i]$ is a Euclidean function. Hint: Consider the exact quotient of two Gaussian integers $\alpha, \beta \in \mathbb{Z}[i]$ in $\mathbb{C}$.

(ii) Show that the units in $\mathbb{Z}[i]$ are precisely the elements of norm 1 and enumerate them.

(iii) Prove that there is no multiplicative normal form on $\mathbb{Z}[i]$ which extends the usual normal form $\text{normal}(a) = |a|$ on $\mathbb{Z}$. Hint: Consider $\text{normal}((1+i)^2)$. Why is $\text{normal}(a+ib) = |a| + i|b|$ for $a, b \in \mathbb{Z}$ *not* a normal form?

(iv) Compute all greatest common divisors of 6 and $3 + i$ in $\mathbb{Z}[i]$ and their representations as a linear combination of 6 and $3 + i$.

(v) Compute a gcd of 12 277 and $399 + 20i$.

3.20* Let $x_1, x_2, \ldots$ be countably many indeterminates over $\mathbb{Z}$, $R = \mathbb{Z}[x_1, x_2, \ldots]$,

$$Q_i = \begin{pmatrix} 0 & 1 \\ 1 & x_i \end{pmatrix} \in R^{2\times 2}$$

for $i \geq 1$, and $R_i = Q_i \cdots Q_1$. We define the ith **continuant polynomial** $c_i \in R$ recursively by $c_0 = 0$, $c_1 = 1$, and $c_{i+1} = c_{i-1} + x_i c_i$ for $i \geq 1$. Then $c_i \in \mathbb{Z}[x_1, \ldots, x_{i-1}]$ for $i \geq 1$.

(i) List the first 10 continuant polynomials.

(ii) Let T be the "shift homomorphism" $Tx_i = x_{i+1}$ for $i \geq 1$. Show that $c_{i+2}(0, x_1, x_2, \ldots, x_i) = Tc_i$ for $i \geq 0$.

(iii) Show that $R_i = \begin{pmatrix} Tc_{i-1} & c_i \\ T_{c_i} & c_{i+1} \end{pmatrix}$ for $i \geq 1$.

(iv) Show that $\det R_i = (-1)^i$, and conclude that $\gcd(c_i, c_{i+1}) = 1$ for $i \geq 0$.

(v) Let D be a Euclidean domain and $r_i, q_i, s_i, t_i \in D$ for $0 \leq i \leq \ell$ the results of the *traditional* Extended Euclidean Algorithm for r_0, r_1. Show that

$$\begin{aligned} s_i &= c_{i-1}(-q_2, \ldots, -q_{i-1}) = (-1)^i c_{i-1}(q_2, \ldots, q_{i-1}), \\ t_i &= c_i(-q_1, \ldots, -q_{i-1}) = (-1)^{i-1} c_i(q_1, \ldots, q_{i-1}) \end{aligned}$$

for $1 \leq i \leq \ell$.

(vi) Write a MAPLE program that implements the traditional Extended Euclidean Algorithm and additionally computes all continuants $c_i(q_{\ell-i+2}, \ldots, q_\ell)$ for $r_0 = x^{20}$ and $r_1 = x^{19} + 2x^{18} + x$ in $\mathbb{Q}[x]$, where $q_1, \ldots, q_\ell$ are the quotients in the traditional Extended Euclidean Algorithm.

3.21 (i) Prove Lemma 3.10 (6) for the traditional EEA 3.6. Hint: Since q_1 may be constant, it is wise to start the induction with $i = 3$ and show the cases $i = 1$ and $i = 2$ separately.

(ii) Prove Lemma 3.10 for the Extended Euclidean Algorithm 3.14.

3.22 (i) Show that for polynomials $f, g \in F[x]$ of degrees $n \geq m$, where F is a field, computing all entries s_i in the traditional Extended Euclidean Algorithm from the quotients q_i takes at most $2m^2 + 2m$ additions and multiplications in F. Hint: Exhibit the bound for the normal case and prove that this is the worst case.

(ii) Prove that the corresponding estimate for the Extended Euclidean Algorithm is $2m^2 + 3m + 1$.

3.23 Prove Lemma 3.12. Hint: Use Lemma 3.8 and Exercise 3.15.

3.24* Prove Theorem 3.13.

3.25* We consider the following recursive algorithm for computing the gcd of two integers.

ALGORITHM 3.17 Binary Euclidean Algorithm.

Input: $a, b \in \mathbb{N}_{>0}$.

Output: $\gcd(a, b) \in \mathbb{N}$.

1. **if** $a = b$ **then return** a
2. **if** both a and b are even **then return** $2 \cdot \gcd(a/2, b/2)$
3. **if** exactly one of the two numbers, say a, is even **then return** $\gcd(a/2, b)$
4. **if** both a and b are odd and, say, $a > b$, **then return** $\gcd((a-b)/2, b)$

(i) Run the algorithm on the examples of Exercise 3.11.

(ii) Prove that the algorithm works correctly.

(iii) Find a "good" upper bound on the recursion depth of the algorithm, and show that it takes $O(n^2)$ word operations on inputs of length at most n.

(iv) Modify the algorithm so that it additionally computes $s, t \in \mathbb{N}$ such that $sa + tb = \gcd(a, b)$.

3.26* Adapt the algorithm from Exercise 3.25 to polynomials over a field. Hint: Start with $\mathbb{F}_2[x]$.

3.27 Let F_n and F_{n+1} be consecutive terms in the Fibonacci sequence. Show that $\gcd(F_{n+1}, F_n) = 1$.

3.28 (i) Prove the formula

$$F_n = \frac{1}{\sqrt{5}}(\phi_+^n - \phi_-^n) \text{ for } n \in \mathbb{N} \tag{10}$$

for the Fibonacci numbers, where $\phi_+ = (1+\sqrt{5})/2 \approx 1.618$ is the golden ratio and $\phi_- = -1/\phi_+ = (1-\sqrt{5})/2 \approx -0.618$. Conclude that F_n is the nearest integer to $\phi_+^n/\sqrt{5}$ for all n.

(ii) For $n \in \mathbb{N}_{>0}$, let $k_n = [1, \ldots, 1]$ be the continued fraction of length n with all entries equal to 1 (Section 4.6). Prove that $k_n = F_{n+1}/F_n$, and conclude that $\lim_{n \longrightarrow \infty} k_n = \phi_+$.

3.29* This continues Exercise 3.28.

(i) Let $h = \sum_{n \geq 0} F_n x^n \in \mathbb{Q}[[x]]$ be the formal power series whose coefficients are the Fibonacci numbers. Derive a linear equation for h from the recursion formula for the Fibonacci numbers and solve it for h. (It will turn out that h is a rational function in x.)

(ii) Compute the partial fraction expansion (Section 5.11) of h and use it to prove (10) again by employing the formula $\sum_{n \geq 0} x^n = 1/(1-x)$ for the geometric series and comparing coefficients.

3.30* In the least absolute remainder variant of the Euclidean Algorithm for integers (Exercise 3.13), all quotients q_i (with the possible exception of q_1) are at least two in absolute value. Thus the nonnegative integers with the largest possible number of division steps in this variant, that is, the analog of the Fibonacci numbers in Lamé's theorem, are recursively defined by

$$G_0 = 0, \quad G_1 = 1, \quad G_{n+1} = 2G_n + G_{n-1} \text{ for } n \geq 1.$$

(i) Find a closed form expression similar to (10) for G_n. Hint: Proceed as in Exercise 3.29.

(ii) Derive a tight upper bound on the length ℓ of the least absolute remainder Euclidean Algorithm for two integers $f, g \in \mathbb{N}$ with $f > g$ in terms of $\log g$, and compare it to (8).

3.31* For $n \in \mathbb{N}$, let F_n be the nth Fibonacci number, with $F_0 = 0$ and $F_1 = 1$. Prove or disprove that the following properties hold for all $n, k \in \mathbb{N}$.

(i) $F_{n+k+1} = F_n F_k + F_{n+1} F_{k+1}$,

(ii) F_k divides F_{nk},

(iii) $\gcd(F_{nk+1}, F_k) = 1$ if $k \geq 1$ (hint: Exercise 3.27),

(iv) $F_n \operatorname{rem} F_k = F_{n \operatorname{rem} k}$ if $k \geq 1$,

(v) $\gcd(F_n, F_k) = \gcd(F_k, F_{n \operatorname{rem} k})$ if $k \geq 1$ (hint: Exercise 3.16),

(vi) $\gcd(F_n, F_k) = F_{\gcd(n,k)}$.

(vii) Conclude from (i) that F_n can be calculated with $O(\log n)$ arithmetic operations in $\mathbb{Z}$.

(viii) Generalize your answers to **Lucas sequences** $(L_n)_{n \geq 0}$ of the form $L_0 = 0, L_1 = 1$, and $L_{n+2} = aL_{n+1} + L_n$ for $n \in \mathbb{N}$, where $a \in \mathbb{Z}$ is a fixed constant.

3.32* We define the sequence $f_0, f_1, f_2, \ldots \in \mathbb{Q}[x]$ of monic polynomials by

- $\gcd(f_n, f_{n-1}) = 1$ for $n \geq 1$,
- for every $n \geq 1$ the number of division steps in the Euclidean Algorithm for (f_n, f_{n-1}) is n, and all quotients are equal to x.

(i) What are the remainders in the Euclidean Algorithm for f_n and f_{n-1}? What are the ρ_i? Find a recursion for the f_n. What is the degree of f_n?

(ii) What is the connection between the f_n and the Fibonacci numbers?

(iii) State and prove a theorem saying that the number of division steps in the Euclidean Algorithm for the pair (f_n, f_{n-1}) is maximal. Make explicit what you mean by maximal.

3.33 Let R be a ring, and $f, g, q, r \in R[x]$ with $g \neq 0$, $f = qg + r$, and $\deg r < \deg g$. Prove that q and r are unique if and only if $\operatorname{lc}(g)$ is not a zero divisor.