Henri Cohen

A Course in Computational Algebraic Number Theory

Springer

Henri Cohen
U.F.R. de Mathématiques et Informatique
Université Bordeaux I
351 Cours de la Libération
F-33405 Talence Cedex, France

Third, Corrected Printing 1996
With 1 Figure

Mathematics Subject Classification (1991): 11Y05, 11Y11, 11Y16, 11Y40, 11A51, 11C08, 11C20, 11R09, 11R11, 11R29

ISSN 0072-5285
ISBN 3-540-55640-0 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-55640-0 Springer-Verlag New York Berlin Heidelberg

Cataloging-In-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Cohen, Henri:
A course in computational algebraic number theory / Henri Cohen. - 3., corr. print. - Berlin ; Heidelberg ; New York : Springer, 1996
(Graduate texts in mathematics; 138)
ISBN 3-540-55640-0
NE: GT

Typesetting: Camera-ready copy produced from the author's output file using A_MS-TEX and LA_MS-TEX
SPIN: 10558047 41/3143-5 4 3 2 1 0 – Printed on acid-free paper

Acknowledgments

This book grew from notes prepared for graduate courses in computational number theory given at the University of Bordeaux I. When preparing this book, it seemed natural to include both more details and more advanced subjects than could be given in such a course. By doing this, I hope that the book can serve two audiences: the mathematician who might only need the details of certain algorithms as well as the mathematician wanting to go further with algorithmic number theory.

In 1991, we started a graduate program in computational number theory in Bordeaux, and this book was also meant to provide a framework for future courses in this area.

In roughly chronological order I need to thank, Horst Zimmer, whose Springer Lecture Notes on the subject [Zim] was both a source of inspiration and of excellent references for many people at the time when it was published.

Then, certainly, thanks must go to Donald Knuth, whose (unfortunately unfinished) series on the Art of Computer Programming ([Knu1], [Knu2] and [Knu3]) contains many marvels for a mathematician. In particular, the second edition of his second volume. Parts of the contents of Chapters 1 and 3 of this book are taken with little or no modifications from Knuth's book. In the (very rare) cases where Knuth goes wrong, this is explicitly mentioned.

My thesis advisor and now colleague Jacques Martinet, has been very influential, both in developing the subject in Bordeaux and more generally in the rest of France–several of his former students are now professors. He also helped to make me aware of the beauty of the subject, since my personal inclination was more towards analytic aspects of number theory, like modular forms or L-functions. Even during the strenuous period (for him!) when he was Chairman of our department, he always took the time to listen or enthusiastically explain.

I also want to thank Hendrik Lenstra, with whom I have had the pleasure of writing a few joint papers in this area. Also Arjen Lenstra, who took the trouble of debugging and improving a big Pascal program which I wrote, which is still, in practice, one of the fastest primality proving programs. Together and separately they have contributed many extremely important algorithms, in particular LLL and its applications (see Section 2.6). My only regret is that they both are now in the U.S.A., so collaboration is more difficult.

Although he is not strictly speaking in the algorithmic field, I must also thank Don Zagier, first for his personal and mathematical friendship and also for his continuing invitations first to Maryland, then at the Max Planck Institute in Bonn, but also because he is a mathematician who takes both real pleasure and real interest in creating or using algorithmic tools in number theory. In fact, we are currently finishing a large algorithmic project, jointly with Nils Skoruppa.

Daniel Shanks[1], both as an author and as editor of Mathematics of Computation, has also had a great influence on the development of algorithmic algebraic number theory. I have had the pleasure of collaborating with him during my 1982 stay at the University of Maryland, and then in a few subsequent meetings.

My colleagues Christian Batut, Dominique Bernardi and Michel Olivier need to be especially thanked for the enormous amount of unrewarding work that they put in the writing of the PARI system under my supervision. This system is now completely operational (even though a few unavoidable bugs crop up from time to time), and is extremely useful for us in Bordeaux, and for the (many) people who have a copy of it elsewhere. It has been and continues to be a great pleasure to work with them.

I also thank my colleague François Dress for having collaborated with me to write our first multi-precision interpreter ISABELLE, which, although considerably less ambitious than PARI, was a useful first step.

I met Johannes Buchmann several years ago at an international meeting. Thanks to the administrative work of Jacques Martinet on the French side, we now have a bilateral agreement between Bordeaux and Saarbrücken. This has allowed several visits, and a medium term joint research plan has been informally decided upon. Special thanks are also due to Johannes Buchmann and Horst Zimmer for this. I need to thank Johannes Buchmann for the many algorithms and techniques which I have learned from him both in published work and in his preprints. A large part of this book could not have been what it is without his direct or indirect help. Of course, I take complete responsibility for the errors that may have appeared!

Although I have met Michael Pohst and Hans Zassenhaus[2] only in meetings and did not have the opportunity to work with them directly, they have greatly influenced the development of modern methods in algorithmic number theory. They have written a book [Poh-Zas] which is a landmark in the subject. I recommend it heartily for further reading, since it goes into subjects which could not be covered in this book.

I have benefited from discussions with many other people on computational number theory, which in alphabetical order are, Oliver Atkin, Anne-Marie Bergé, Bryan Birch, Francisco Diaz y Diaz, Philippe Flajolet, Guy Henniart, Kevin McCurley, Jean-François Mestre, François Morain, Jean-Louis

[1] Daniel Shanks died on September 6, 1996.
[2] Hans Zassenhaus died on November 21, 1991.

Nicolas, Andrew Odlyzko, Joseph Oesterlé, Johannes Graf von Schmettow, Claus-Peter Schnorr, Rene Schoof, Jean-Pierre Serre, Bob Silverman, Harold Stark, Nelson Stephens, Larry Washington. There are many others that could not be listed here. I have taken the liberty of borrowing some of their algorithms, and I hope that I will be forgiven if their names are not always mentioned.

The theoretical as well as practical developments in Computational Number Theory which have taken place in the last few years in Bordeaux would probably not have been possible without a large amount of paperwork and financial support. Hence, special thanks go to the people who made this possible, and in particular to Jean-Marc Deshouillers, François Dress and Jacques Martinet as well as the relevant local and national funding committees and agencies.

I must thank a number of persons without whose help we would have been essentially incapable of using our workstations, in particular "Achille" Braquelaire, Laurent Fallot, Patrick Henry, Viviane Sauquet-Deletage, Robert Strandh and Bernard Vauquelin.

Although I do not know anybody there, I would also like to thank the GNU project and its creator Richard Stallman, for the excellent software they produce, which is not only free (as in "freedom", but also as in "freeware"), but is generally superior to commercial products. Most of the software that we use comes from GNU.

Finally, I thank all the people, too numerous to mention, who have helped me in some way or another to improve the quality of this book, and in particular to Dominique Bernardi and Don Zagier who very carefully read drafts of this book. But special thanks go to Gary Cornell who suggested improvements to my English style and grammar in almost every line.

In addition, several people contributed directly or helped me write specific sections of the book. In alphabetical order they are D. Bernardi (algorithms on elliptic curves), J. Buchmann (Hermite normal forms and sub-exponential algorithms), J.-M. Couveignes (number field sieve), H. W. Lenstra (in several sections and exercises), C. Pomerance (factoring and primality testing), B. Vallée (LLL algorithms), P. Zimmermann (Appendix A).

Preface

With the advent of powerful computing tools and numerous advances in mathematics, computer science and cryptography, algorithmic number theory has become an important subject in its own right. Both external and internal pressures gave a powerful impetus to the development of more powerful algorithms. These in turn led to a large number of spectacular breakthroughs. To mention but a few, the LLL algorithm which has a wide range of applications, including real world applications to integer programming, primality testing and factoring algorithms, sub-exponential class group and regulator algorithms, etc ...

Several books exist which treat parts of this subject. (It is essentially impossible for an author to keep up with the rapid pace of progress in all areas of this subject.) Each book emphasizes a different area, corresponding to the author's tastes and interests. The most famous, but unfortunately the oldest, is Knuth's *Art of Computer Programming*, especially Chapter 4.

The present book has two goals. First, to give a reasonably comprehensive **introductory** course in computational number theory. In particular, although we study some subjects in great detail, others are only mentioned, but with suitable pointers to the literature. Hence, we hope that this book can serve as a first course on the subject. A natural sequel would be to study more specialized subjects in the existing literature.

The prerequisites for reading this book are contained in introductory texts in number theory such as Hardy and Wright [H-W] and Borevitch and Shafarevitch [Bo-Sh]. The reader also needs some feeling or taste for algorithms and their implementation. To make the book as self-contained as possible, the main definitions are given when necessary. However, it would be more reasonable for the reader to first acquire some basic knowledge of the subject before studying the algorithmic part. On the other hand, algorithms often give natural proofs of important results, and this nicely complements the more theoretical proofs which may be given in other books.

The second goal of this course is **practicality**. The author's primary intentions were not only to give fundamental and interesting algorithms, but also to concentrate on practical aspects of the implementation of these algorithms. Indeed, the theory of algorithms being not only fascinating but rich, can be (somewhat arbitrarily) split up into four closely related parts. The first is the discovery of new algorithms to solve particular problems. The second is the detailed mathematical analysis of these algorithms. This is usually quite

mathematical in nature, and quite often intractable, although the algorithms seem to perform rather well in practice. The third task is to study the complexity of the problem. This is where notions of fundamental importance in complexity theory such as NP-completeness come in. The last task, which some may consider the least noble of the four, is to actually implement the algorithms. But this task is of course as essential as the others for the actual resolution of the problem.

In this book we give the algorithms, the mathematical analysis and in some cases the complexity, without proofs in some cases, especially when it suffices to look at the existing literature such as Knuth's book. On the other hand, we have usually tried as carefully as we could, to give the algorithms in a ready to program form–in as optimized a form as possible. This has the drawback that some algorithms are unnecessarily clumsy (this is unavoidable if one optimizes), but has the great advantage that a casual user of these algorithms can simply take them as written and program them in his/her favorite programming language. In fact, the author himself has implemented almost all the algorithms of this book in the number theory package PARI (see Appendix A).

The approach used here as well as the style of presentation of the algorithms is similar to that of Knuth (analysis of algorithms excepted), and is also similar in spirit to the book of Press et al [PFTV] *Numerical Recipes (in Fortran, Pascal or C)*, although the subject matter is completely different.

For the practicality criterion to be compatible with a book of reasonable size, some compromises had to be made. In particular, on the mathematical side, many proofs are not given, especially when they can easily be found in the literature. From the computer science side, essentially no complexity results are proved, although the important ones are stated.

The book is organized as follows. The first chapter gives the fundamental algorithms that are constantly used in number theory, in particular algorithms connected with powering modulo N and with the Euclidean algorithm.

Many number-theoretic problems require algorithms from linear algebra over a field or over $\mathbb{Z}$. This is the subject matter of Chapter 2. The highlights of this chapter are the Hermite and Smith normal forms, and the fundamental LLL algorithm.

In Chapter 3 we explain in great detail the Berlekamp-Cantor-Zassenhaus methods used to factor polynomials over finite fields and over $\mathbb{Q}$, and we also give an algorithm for finding all the complex roots of a polynomial.

Chapter 4 gives an introduction to the algorithmic techniques used in number fields, and the basic definitions and results about algebraic numbers and number fields. The highlights of these chapters are the use of the Hermite Normal Form representation of modules and ideals, an algorithm due to Diaz y Diaz and the author for finding "simple" polynomials defining a number field, and the subfield and field isomorphism problems.

Quadratic fields provide an excellent testing and training ground for the techniques of algorithmic number theory (and for algebraic number theory in general). This is because although they can easily be generated, many non-trivial problems exist, most of which are unsolved (are there infinitely many real quadratic fields with class number 1?). They are studied in great detail in Chapter 5. In particular, this chapter includes recent advances on the efficient computation in class groups of quadratic fields (Shanks's NUCOMP as modified by Atkin), and sub-exponential algorithms for computing class groups and regulators of quadratic fields (McCurley-Hafner, Buchmann).

Chapter 6 studies more advanced topics in computational algebraic number theory. We first give an efficient algorithm for computing integral bases in number fields (Zassenhaus's round 2 algorithm), and a related algorithm which allows us to compute explicitly prime decompositions in field extensions as well as valuations of elements and ideals at prime ideals. Then, for number fields of degree less than or equal to 7 we give detailed algorithms for computing the Galois group of the Galois closure. We also study in some detail certain classes of cubic fields. This chapter concludes with a general algorithm for computing class groups and units in general number fields. This is a generalization of the sub-exponential algorithms of Chapter 5, and works quite well. For other approaches, I refer to [Poh-Zas] and to a forthcoming paper of J. Buchmann. This subject is quite involved so, unlike most other situations in this book, I have not attempted to give an efficient algorithm, just one which works reasonably well in practice.

Chapters 1 to 6 may be thought of as one unit and describe many of the most interesting aspects of the theory. These chapters are suitable for a two semester graduate (or even a senior undergraduate) level course in number theory. Chapter 6, and in particular the class group and unit algorithm, can certainly be considered as a climax of the first part of this book.

A number theorist, especially in the algorithmic field, must have a minimum knowledge of elliptic curves. This is the subject of chapter 7. Excellent books exist about elliptic curves (for example [Sil] and [Sil3]), but our aim is a little different since we are primarily concerned with applications of elliptic curves. But a minimum amount of culture is also necessary, and so the flavor of this chapter is quite different from the others chapters. In the first three sections, we give the essential definitions, and we give the basic and most striking results of the theory, with no pretense to completeness and no algorithms.

The theory of elliptic curves is one of the most marvelous mathematical theories of the twentieth century, and abounds with important conjectures. They are also mentioned in these sections. The last sections of Chapter 7, give a number of useful algorithms for working on elliptic curves, with little or no proofs.

The reader is warned that, apart from the material necessary for later chapters, Chapter 7 needs a much higher mathematical background than the other chapters. It can be skipped if necessary without impairing the understanding of the subsequent chapters.

Chapter 8 (whose title is borrowed from a talk of Hendrik Lenstra) considers the techniques used for primality testing and factoring prior to the 1970's, with the exception of the continued fraction method of Brillhart-Morrison which belongs in Chapter 10.

Chapter 9 explains the theory and practice of the two modern primality testing algorithms, the Adleman-Pomerance-Rumely test as modified by H. W. Lenstra and the author, which uses Fermat's (little) theorem in cyclotomic fields, and Atkin's test which uses elliptic curves with complex multiplication.

Chapter 10 is devoted to modern factoring methods, i.e. those which run in sub-exponential time, and in particular to the Elliptic Curve Method of Lenstra, the Multiple Polynomial Quadratic Sieve of Pomerance and the Number Field Sieve of Pollard. Since many of the methods described in Chapters 9 and 10 are quite complex, it is not reasonable to give ready-to-program algorithms as in the preceding chapters, and the implementation of any one of these complex methods can form the subject of a three month student project.

In Appendix A, we describe what a serious user should know about computer packages for number theory. The reader should keep in mind that the author of this book is biased since he has written such a package himself (this package being available without cost by anonymous ftp).

Appendix B has a number of tables which we think may useful to the reader. For example, they can be used to check the correctness of the implementation of certain algorithms.

What I have tried to cover in this book is so large a subject that, necessarily, it cannot be treated in as much detail as I would have liked. For further reading, I suggest the following books.

For Chapters 1 and 3, [Knu1] and [Knu2]. This is the bible for algorithm analysis. Note that the sections on primality testing and factoring are outdated. Also, algorithms like the LLL algorithm which did not exist at the time he wrote are, obviously, not mentioned. The recent book [GCL] contains essentially all of our Chapter 3, as well as many more polynomial algorithms which we have not covered in this book such as Gröbner bases computation.

For Chapters 4 and 5, [Bo-Sh], [Mar] and [Ire-Ros]. In particular, [Mar] and [Ire-Ros] contain a large number of practical exercises, which are not far from the spirit of the present book, [Ire-Ros] being more advanced.

For Chapter 6, [Poh-Zas] contains a large number of algorithms, and treats in great detail the question of computing units and class groups in general number fields. Unfortunately the presentation is sometimes obscured by quite complicated notations, and a lot of work is often needed to implement the algorithms given there.

For Chapter 7, [Sil] and [Sil3] are excellent books, and contain numerous exercises. Another good reference is [Hus], as well as [Ire-Ros] for material on zeta-functions of varieties. The algorithmic aspect of elliptic curves is beautifully treated in [Cre], which I also heartily recommend.

For Chapters 8 to 10, the best reference to date, in addition to [Knu2], is [Rie]. In addition, Riesel has several chapters on prime number theory.

Note on the exercises. The exercises have a wide range of difficulty, from extremely easy to unsolved research problems. Many are actually implementation problems, and hence not mathematical in nature. No attempt has been made to grade the level of difficulty of the exercises as in Knuth, except of course that unsolved problems are mentioned as such. The ordering follows roughly the corresponding material in the text.

WARNING. Almost all of the algorithms given in this book have been programmed by the author and colleagues, in particular as a part of the Pari package. The programming has not however, always been synchronized with the writing of this book, so it may be that some algorithms are incorrect, and others may contain slight typographical errors which of course also invalidate them. Hence, the author and Springer-Verlag do not assume any responsibility for consequences which may directly or indirectly occur from the use of the algorithms given in this book. Apart from the preceding legalese, the author would appreciate corrections, improvements and so forth to the algorithms given, so that this book may improve if further editions are printed. The simplest is to send an e-mail message to

`cohen@math.u-bordeaux.fr`

or else to write to the author's address. In addition, a regularly updated errata file is available by anonymous ftp from `megrez.math.u-bordeaux.fr` (`147.210.16.17`), directory `pub/cohenbook`.

Contents

Chapter 1 Fundamental Number-Theoretic Algorithms 1

1.1 Introduction 1

1.1.1 Algorithms 1
1.1.2 Multi-precision 2
1.1.3 Base Fields and Rings 5
1.1.4 Notations 6

1.2 The Powering Algorithms 8

1.3 Euclid's Algorithms 12

1.3.1 Euclid's and Lehmer's Algorithms 12
1.3.2 Euclid's Extended Algorithms 16
1.3.3 The Chinese Remainder Theorem 19
1.3.4 Continued Fraction Expansions of Real Numbers 21

1.4 The Legendre Symbol 24

1.4.1 The Groups $(\mathbb{Z}/n\mathbb{Z})^*$ 24
1.4.2 The Legendre-Jacobi-Kronecker Symbol 27

1.5 Computing Square Roots Modulo p 31

1.5.1 The Algorithm of Tonelli and Shanks 32
1.5.2 The Algorithm of Cornacchia 34

1.6 Solving Polynomial Equations Modulo p 36

1.7 Power Detection 38

1.7.1 Integer Square Roots 38
1.7.2 Square Detection 39
1.7.3 Prime Power Detection 41

1.8 Exercises for Chapter 1 42

Chapter 2 Algorithms for Linear Algebra and Lattices 46

2.1 Introduction 46

2.2 Linear Algebra Algorithms on Square Matrices 47

2.2.1 Generalities on Linear Algebra Algorithms 47
2.2.2 Gaussian Elimination and Solving Linear Systems 48
2.2.3 Computing Determinants 50
2.2.4 Computing the Characteristic Polynomial 53

2.3 Linear Algebra on General Matrices 57

2.3.1 Kernel and Image 57
2.3.2 Inverse Image and Supplement 60
2.3.3 Operations on Subspaces 62
2.3.4 Remarks on Modules 64

2.4 $\mathbb{Z}$-Modules and the Hermite and Smith Normal Forms . . 66

2.4.1 Introduction to $\mathbb{Z}$-Modules 66
2.4.2 The Hermite Normal Form 67
2.4.3 Applications of the Hermite Normal Form 73
2.4.4 The Smith Normal Form and Applications 75

2.5 Generalities on Lattices 79

2.5.1 Lattices and Quadratic Forms 79
2.5.2 The Gram-Schmidt Orthogonalization Procedure 82

2.6 Lattice Reduction Algorithms 84

2.6.1 The LLL Algorithm 84
2.6.2 The LLL Algorithm with Deep Insertions 90
2.6.3 The Integral LLL Algorithm 92
2.6.4 LLL Algorithms for Linearly Dependent Vectors 95

2.7 Applications of the LLL Algorithm 97

2.7.1 Computing the Integer Kernel and Image of a Matrix 97
2.7.2 Linear and Algebraic Dependence Using LLL 100
2.7.3 Finding Small Vectors in Lattices 103

2.8 Exercises for Chapter 2 106

Chapter 3 Algorithms on Polynomials 109

3.1 Basic Algorithms 109

3.1.1 Representation of Polynomials 109
3.1.2 Multiplication of Polynomials 110
3.1.3 Division of Polynomials 111

3.2 Euclid's Algorithms for Polynomials 113

3.2.1 Polynomials over a Field 113
3.2.2 Unique Factorization Domains (UFD's) 114
3.2.3 Polynomials over Unique Factorization Domains 116

3.2.4 Euclid's Algorithm for Polynomials over a UFD 117

3.3 The Sub-Resultant Algorithm 118

3.3.1 Description of the Algorithm 118
3.3.2 Resultants and Discriminants 119
3.3.3 Resultants over a Non-Exact Domain 123

3.4 Factorization of Polynomials Modulo p 124

3.4.1 General Strategy 124
3.4.2 Squarefree Factorization 125
3.4.3 Distinct Degree Factorization 126
3.4.4 Final Splitting 127

3.5 Factorization of Polynomials over $\mathbb{Z}$ or $\mathbb{Q}$ 133

3.5.1 Bounds on Polynomial Factors 134
3.5.2 A First Approach to Factoring over $\mathbb{Z}$ 135
3.5.3 Factorization Modulo p^e: Hensel's Lemma 137
3.5.4 Factorization of Polynomials over $\mathbb{Z}$ 139
3.5.5 Discussion 141

3.6 Additional Polynomial Algorithms 142

3.6.1 Modular Methods for Computing GCD's in $\mathbb{Z}[X]$ 142
3.6.2 Factorization of Polynomials over a Number Field 143
3.6.3 A Root Finding Algorithm over $\mathbb{C}$ 146

3.7 Exercises for Chapter 3 148

Chapter 4 Algorithms for Algebraic Number Theory I 153

4.1 Algebraic Numbers and Number Fields 153

4.1.1 Basic Definitions and Properties of Algebraic Numbers 153
4.1.2 Number Fields 154

4.2 Representation and Operations on Algebraic Numbers . . 158

4.2.1 Algebraic Numbers as Roots of their Minimal Polynomial 158
4.2.2 The Standard Representation of an Algebraic Number 159
4.2.3 The Matrix (or Regular) Representation of an Algebraic Number . 160
4.2.4 The Conjugate Vector Representation of an Algebraic Number . . 161

4.3 Trace, Norm and Characteristic Polynomial 162

4.4 Discriminants, Integral Bases and Polynomial Reduction . 165

4.4.1 Discriminants and Integral Bases 165
4.4.2 The Polynomial Reduction Algorithm 168

4.5 The Subfield Problem and Applications 174

4.5.1 The Subfield Problem Using the LLL Algorithm 174
4.5.2 The Subfield Problem Using Linear Algebra over $\mathbb{C}$ 175
4.5.3 The Subfield Problem Using Algebraic Algorithms 177
4.5.4 Applications of the Solutions to the Subfield Problem 179

4.6 Orders and Ideals . . . 181
4.6.1 Basic Definitions . . . 181
4.6.2 Ideals of $\mathbb{Z}_K$ . . . 186

4.7 Representation of Modules and Ideals . . . 188
4.7.1 Modules and the Hermite Normal Form . . . 188
4.7.2 Representation of Ideals . . . 190

4.8 Decomposition of Prime Numbers I . . . 196
4.8.1 Definitions and Main Results . . . 196
4.8.2 A Simple Algorithm for the Decomposition of Primes . . . 199
4.8.3 Computing Valuations . . . 201
4.8.4 Ideal Inversion and the Different . . . 204

4.9 Units and Ideal Classes . . . 207
4.9.1 The Class Group . . . 207
4.9.2 Units and the Regulator . . . 209
4.9.3 Conclusion: the Main Computational Tasks of Algebraic Number Theory . . . 217

4.10 Exercises for Chapter 4 . . . 217

Chapter 5 Algorithms for Quadratic Fields . . . 223

5.1 Discriminant, Integral Basis and Decomposition of Primes 223

5.2 Ideals and Quadratic Forms . . . 225

5.3 Class Numbers of Imaginary Quadratic Fields . . . 231
5.3.1 Computing Class Numbers Using Reduced Forms . . . 231
5.3.2 Computing Class Numbers Using Modular Forms . . . 234
5.3.3 Computing Class Numbers Using Analytic Formulas . . . 237

5.4 Class Groups of Imaginary Quadratic Fields . . . 240
5.4.1 Shanks's Baby Step Giant Step Method . . . 240
5.4.2 Reduction and Composition of Quadratic Forms . . . 243
5.4.3 Class Groups Using Shanks's Method . . . 250

5.5 McCurley's Sub-exponential Algorithm . . . 252
5.5.1 Outline of the Algorithm . . . 252
5.5.2 Detailed Description of the Algorithm . . . 255
5.5.3 Atkin's Variant . . . 260

5.6 Class Groups of Real Quadratic Fields . . . 262
5.6.1 Computing Class Numbers Using Reduced Forms . . . 262
5.6.2 Computing Class Numbers Using Analytic Formulas . . . 266
5.6.3 A Heuristic Method of Shanks . . . 268

5.7 Computation of the Fundamental Unit and of the Regulator . . . 269
5.7.1 Description of the Algorithms . . . 269
5.7.2 Analysis of the Continued Fraction Algorithm . . . 271
5.7.3 Computation of the Regulator . . . 278
5.8 The Infrastructure Method of Shanks . . . 279
5.8.1 The Distance Function . . . 279
5.8.2 Description of the Algorithm . . . 283
5.8.3 Compact Representation of the Fundamental Unit . . . 285
5.8.4 Other Application and Generalization of the Distance Function . 287
5.9 Buchmann's Sub-exponential Algorithm . . . 288
5.9.1 Outline of the Algorithm . . . 289
5.9.2 Detailed Description of Buchmann's Sub-exponential Algorithm . 291
5.10 The Cohen-Lenstra Heuristics . . . 295
5.10.1 Results and Heuristics for Imaginary Quadratic Fields . . . 295
5.10.2 Results and Heuristics for Real Quadratic Fields . . . 297
5.11 Exercises for Chapter 5 . . . 298

Chapter 6 Algorithms for Algebraic Number Theory II 303
6.1 Computing the Maximal Order . . . 303
6.1.1 The Pohst-Zassenhaus Theorem . . . 303
6.1.2 The Dedekind Criterion . . . 305
6.1.3 Outline of the Round 2 Algorithm . . . 308
6.1.4 Detailed Description of the Round 2 Algorithm . . . 311
6.2 Decomposition of Prime Numbers II . . . 312
6.2.1 Newton Polygons . . . 313
6.2.2 Theoretical Description of the Buchmann-Lenstra Method . . . 315
6.2.3 Multiplying and Dividing Ideals Modulo p . . . 317
6.2.4 Splitting of Separable Algebras over $\mathbb{F}_p$ . . . 318
6.2.5 Detailed Description of the Algorithm for Prime Decomposition . 320
6.3 Computing Galois Groups . . . 322
6.3.1 The Resolvent Method . . . 322
6.3.2 Degree 3 . . . 325
6.3.3 Degree 4 . . . 325
6.3.4 Degree 5 . . . 328
6.3.5 Degree 6 . . . 329
6.3.6 Degree 7 . . . 331
6.3.7 A List of Test Polynomials . . . 333
6.4 Examples of Families of Number Fields . . . 334
6.4.1 Making Tables of Number Fields . . . 334
6.4.2 Cyclic Cubic Fields . . . 336

6.4.3 Pure Cubic Fields 343
6.4.4 Decomposition of Primes in Pure Cubic Fields 347
6.4.5 General Cubic Fields 351

6.5 Computing the Class Group, Regulator and Fundamental Units 352
6.5.1 Ideal Reduction 352
6.5.2 Computing the Relation Matrix 354
6.5.3 Computing the Regulator and a System of Fundamental Units . . 357
6.5.4 The General Class Group and Unit Algorithm 358
6.5.5 The Principal Ideal Problem 360

6.6 Exercises for Chapter 6 362

Chapter 7 Introduction to Elliptic Curves 367

7.1 Basic Definitions 367
7.1.1 Introduction 367
7.1.2 Elliptic Integrals and Elliptic Functions 367
7.1.3 Elliptic Curves over a Field 369
7.1.4 Points on Elliptic Curves 372

7.2 Complex Multiplication and Class Numbers 376
7.2.1 Maps Between Complex Elliptic Curves 377
7.2.2 Isogenies 379
7.2.3 Complex Multiplication 381
7.2.4 Complex Multiplication and Hilbert Class Fields 384
7.2.5 Modular Equations 385

7.3 Rank and L-functions 386
7.3.1 The Zeta Function of a Variety 387
7.3.2 L-functions of Elliptic Curves 388
7.3.3 The Taniyama-Weil Conjecture 390
7.3.4 The Birch and Swinnerton-Dyer Conjecture 392

7.4 Algorithms for Elliptic Curves 394
7.4.1 Algorithms for Elliptic Curves over $\mathbb{C}$ 394
7.4.2 Algorithm for Reducing a General Cubic 399
7.4.3 Algorithms for Elliptic Curves over $\mathbb{F}_p$ 403

7.5 Algorithms for Elliptic Curves over $\mathbb{Q}$ 406
7.5.1 Tate's algorithm 406
7.5.2 Computing rational points 410
7.5.3 Algorithms for computing the L-function 413

7.6 Algorithms for Elliptic Curves with Complex Multiplication 414
7.6.1 Computing the Complex Values of $j(\tau)$ 414
7.6.2 Computing the Hilbert Class Polynomials 415

7.6.3 Computing Weber Class Polynomials 416
7.7 Exercises for Chapter 7 417

Chapter 8 Factoring in the Dark Ages 419
8.1 Factoring and Primality Testing 419
8.2 Compositeness Tests 421
8.3 Primality Tests 423
8.3.1 The Pocklington-Lehmer $N-1$ Test 423
8.3.2 Briefly, Other Tests 424
8.4 Lehman's Method 425
8.5 Pollard's ρ Method 426
8.5.1 Outline of the Method 426
8.5.2 Methods for Detecting Periodicity 427
8.5.3 Brent's Modified Algorithm 429
8.5.4 Analysis of the Algorithm 430
8.6 Shanks's Class Group Method 433
8.7 Shanks's SQUFOF 434
8.8 The $p-1$-method 438
8.8.1 The First Stage 439
8.8.2 The Second Stage 440
8.8.3 Other Algorithms of the Same Type 441
8.9 Exercises for Chapter 8 442

Chapter 9 Modern Primality Tests 445
9.1 The Jacobi Sum Test 446
9.1.1 Group Rings of Cyclotomic Extensions 446
9.1.2 Characters, Gauss Sums and Jacobi Sums 448
9.1.3 The Basic Test 450
9.1.4 Checking Condition $\mathcal{L}_p$ 455
9.1.5 The Use of Jacobi Sums 457
9.1.6 Detailed Description of the Algorithm 463
9.1.7 Discussion 465
9.2 The Elliptic Curve Test 467
9.2.1 The Goldwasser-Kilian Test 467
9.2.2 Atkin's Test 471
9.3 Exercises for Chapter 9 475

Chapter 10 Modern Factoring Methods 477

10.1 The Continued Fraction Method 477

10.2 The Class Group Method 481

10.2.1 Sketch of the Method 481
10.2.2 The Schnorr-Lenstra Factoring Method 482

10.3 The Elliptic Curve Method 484

10.3.1 Sketch of the Method 484
10.3.2 Elliptic Curves Modulo N 485
10.3.3 The ECM Factoring Method of Lenstra 487
10.3.4 Practical Considerations 489

10.4 The Multiple Polynomial Quadratic Sieve 490

10.4.1 The Basic Quadratic Sieve Algorithm 491
10.4.2 The Multiple Polynomial Quadratic Sieve 492
10.4.3 Improvements to the MPQS Algorithm 494

10.5 The Number Field Sieve 495

10.5.1 Introduction 495
10.5.2 Description of the Special NFS when $h(K) = 1$ 496
10.5.3 Description of the Special NFS when $h(K) > 1$ 500
10.5.4 Description of the General NFS 501
10.5.5 Miscellaneous Improvements to the Number Field Sieve 503

10.6 Exercises for Chapter 10 504

Appendix A Packages for Number Theory 507

Appendix B Some Useful Tables 513

B.1 Table of Class Numbers of Complex Quadratic Fields . . 513

B.2 Table of Class Numbers and Units of Real Quadratic Fields 515

B.3 Table of Class Numbers and Units of Complex Cubic Fields 519

B.4 Table of Class Numbers and Units of Totally Real Cubic Fields 521

B.5 Table of Elliptic Curves 524

Bibliography 527

Index 540

Chapter 1

Fundamental Number-Theoretic Algorithms

1.1 Introduction

This book describes in detail a number of algorithms used in algebraic number theory and the theory of elliptic curves. It also gives applications to problems such as factoring and primality testing. Although the algorithms and the theory behind them are sufficiently interesting in themselves, I strongly advise the reader to take the time to implement them on her/his favorite machine. Indeed, one gets a feel for an algorithm mainly after executing it several times. (This book does help by providing many tricks that will be useful for doing this.)

We give the necessary background on number fields and classical algebraic number theory in Chapter 4, and the necessary prerequisites on elliptic curves in Chapter 7. This chapter shows you some basic algorithms used almost constantly in number theory. The best reference here is [Knu2].

1.1.1 Algorithms

Before we can describe even the simplest algorithms, it is necessary to precisely define a few notions. However, we will do this without entering into the sometimes excessively detailed descriptions used in Computer Science. For us, an *algorithm* will be a method which, given certain types of inputs, gives an answer after a finite amount of time.

Several things must be considered when one describes an algorithm. The first is to prove that it is correct, i.e. that it gives the desired result when it stops. Then, since we are interested in practical implementations, we must give an estimate of the algorithm's running time, if possible both in the worst case, and on average. Here, one must be careful: the running time will always be measured in *bit operations*, i.e. logical or arithmetic operations on zeros and ones. This is the most realistic model, if one assumes that one is using real computers, and not idealized ones. Third, the space requirement (measured in bits) must also be considered. In many algorithms, this is negligible, and then we will not bother mentioning it. In certain algorithms however, it becomes an important issue which has to be addressed.

First, some useful terminology: The size of the inputs for an algorithm will usually be measured by the number of bits that they require. For example, the size of a positive integer N is $\lfloor \lg N \rfloor + 1$ (see below for notations). We

will say that an algorithm is *linear*, *quadratic* or *polynomial time* if it requires time $O(\ln N)$, $O(\ln^2 N)$, $O(P(\ln N))$ respectively, where P is a polynomial. If the time required is $O(N^\alpha)$, we say that the algorithm is exponential time. Finally, many algorithms have some intermediate running time, for example

$$e^{C\sqrt{\ln N \ln\ln N}},$$

which is the approximate expected running time of many factoring algorithms and of recent algorithms for computing class groups. In this case we say that the algorithm is *sub-exponential*.

The definition of algorithm which we have given above, although a little vague, is often still too strict for practical use. We need also *probabilistic algorithms*, which depend on a source of random numbers. These "algorithms" should in principle not be called algorithms since there is a possibility (of probability zero) that they do not terminate. Experience shows, however, that probabilistic algorithms are usually more efficient than non-probabilistic ones; in many cases they are even the only ones available.

Probabilistic algorithms should not be mistaken with methods (which I refuse to call algorithms), which produce a result which has a high probability of being correct. It is essential that an algorithm produces correct results (discounting human or computer errors), even if this happens after a very long time. A typical example of a non-algorithmic method is the following: suppose N is large and you suspect that it is prime (because it is not divisible by small numbers). Then you can compute

$$2^{N-1} \bmod N$$

using the powering Algorithm 1.2.1 below. If it is not $1 \bmod N$, then this proves that N is not prime by Fermat's theorem. On the other hand, if it is equal to $1 \bmod N$, there is a very good chance that N is indeed a prime. But this is not a proof, hence not an algorithm for primality testing (the smallest counterexample is $N = 341$).

Another point to keep in mind for probabilistic algorithms is that the idea of absolute running time no longer makes much sense. This is replaced by the notion of expected running time, which is self-explanatory.

1.1.2 Multi-precision

Since the numbers involved in our algorithms will almost always become quite large, a prerequisite to any implementation is some sort of multi-precision package. This package should be able to handle numbers having up to 1000 decimal digits. Such a package is easy to write, and one is described in detail in Riesel's book ([Rie]). One can also use existing packages or languages, such as Axiom, Bignum, Derive, Gmp, Lisp, Macsyma, Magma, Maple, Mathematica, Pari, Reduce, or Ubasic (see Appendix A). Even without a multi-precision

package, some algorithms can be nicely tested, but their scope becomes more limited.

The pencil and paper method for doing the usual operations can be implemented without difficulty. One should not use a base-10 representation, but rather a base suited to the computer's hardware.

Such a bare-bones multi-precision package must include at the very least:

- Addition and subtraction of two n-bit numbers (time linear in n).
- Multiplication and Euclidean division of two n-bit numbers (time linear in n^2).
- Multiplication and division of an n-bit number by a short integer (time linear in n). Here the meaning of short integer depends on the machine. Usually this means a number of absolute value less than 2^{15}, 2^{31}, 2^{35} or 2^{63}.
- Left and right shifts of an n bit number by small integers (time linear in n).
- Input and output of an n-bit number (time linear in n or in n^2 depending whether the base is a power of 10 or not).

Remark. Contrary to the choice made by some systems such as Maple, I strongly advise using a power of 2 as a base, since usually the time needed for input/output is only a very small part of the total time, and it is also often dominated by the time needed for physical printing or displaying the results.

There exist algorithms for multiplication and division which as n gets large are much faster than $O(n^2)$, the best, due to Schönhage and Strassen, running in $O(n \ln n \ln \ln n)$ bit operations. Since we will be working mostly with numbers of up to roughly 100 decimal digits, it is not worthwhile to implement these more sophisticated algorithms. (These algorithms become practical only for numbers having more than several hundred decimal digits.) On the other hand, simpler schemes such as the method of Karatsuba (see [Knu2] and Exercise 2) can be useful for much smaller numbers.

The times given above for the basic operations should constantly be kept in mind.

Implementation advice. For people who want to write their own bare-bones multi-precision package as described above, by far the best reference is [Knu2] (see also [Rie]). A few words of advice are however necessary. A priori, one can write the package in one's favorite high level language. As will be immediately seen, this limits the multi-precision base to roughly the square root of the word size. For example, on a typical 32 bit machine, a high level language will be able to multiply two 16-bit numbers, but not two 32-bit ones since the result would not fit. Since the multiplication algorithm used is quadratic, this immediately implies a loss of a factor 4, which in fact usually becomes a factor of 8 or 10 compared to what could be done with the machine's central processor. This is intolerable. Another alternative is to write everything in assembly language. This is extremely long and painful, usually

bug-ridden, and in addition not portable, but at least it is fast. This is the solution used in systems such as Pari and Ubasic, which are much faster than their competitors when it comes to pure number crunching.

There is a third possibility which is a reasonable compromise. Declare global variables (known to all the files, including the assembly language files if any) which we will call `remainder` and `overflow` say.

Then write in any way you like (in assembly language or as high level language macros) nine functions that do the following. Assume `a,b,c` are unsigned word-sized variables, and let `M` be the chosen multi-precision base, so all variables will be less than `M` (for example $\texttt{M} = 2^{32}$). Then we need the following functions, where $0 \leq \texttt{c} < \texttt{M}$ and `overflow` is equal to 0 or 1:

`c=add(a,b)` corresponding to the formula `a+b=overflow·M+c`.

`c=addx(a,b)` corresponding to the formula `a+b+overflow=overflow·M+c`.

`c=sub(a,b)` corresponding to the formula `a-b=c-overflow·M`.

`c=subx(a,b)` corresponding to the formula `a-b-overflow=c-overflow·M`.

`c=mul(a,b)` corresponding to the formula `a·b=remainder·M+c`,

in other words `c` contains the low order part of the product, and `remainder` the high order part.

`c=div(a,b)` corresponding to the formula `remainder·M+a=b·c+remainder`, where we may assume that `remainder<b`.

For the last three functions we assume that `M` is equal to a power of 2, say $\texttt{M} = 2^{\texttt{m}}$.

`c=shiftl(a,k)` corresponding to the formula $2^{\texttt{k}}$`a=remainder·M+c`.

`c=shiftr(a,k)` corresponding to the formula $\texttt{a}\cdot\texttt{M}/2^{\texttt{k}}=\texttt{c}\cdot\texttt{M}+\texttt{remainder}$, where we assume for these last two functions that $0 \leq \texttt{k} < \texttt{m}$.

`k=bfffo(a)` corresponding to the formula $\texttt{M}/2 \leq 2^{\texttt{k}}\texttt{a} < \texttt{M}$, i.e. $\texttt{k} = \lceil \lg(\texttt{M}/(2\texttt{a})) \rceil$ when $\texttt{a} \neq 0$, $\texttt{k} = \texttt{m}$ when $\texttt{a} = 0$.

The advantage of this scheme is that the rest of the multi-precision package can be written in a high level language without much sacrifice of speed, and that the black boxes described above are short and easy to write in assembly language. The portability problem also disappears since these functions can easily be rewritten for another machine.

Knowledgeable readers may have noticed that the functions above correspond to a simulation of a few machine language instructions of the 68020/68030/68040 processors. It may be worthwhile to work at a higher level, for example by implementing in assembly language a few of the multi-precision functions mentioned at the beginning of this section. By doing this to a limited extent one can avoid many debugging problems. This also avoids much function call overhead, and allows easier optimizing. As usual, the price paid is portability and robustness.

Remark. One of the most common operations used in number theory is *modular multiplication*, i.e. the computation of $a \cdot b$ modulo some number N, where a and b are non-negative integers less than N. This can, of course,

be trivially done using the formula `div(mul(a,b),N)`, the result being the value of `remainder`. When many such operations are needed using the *same* modulus N (this happens for example in most factoring methods, see Chapters 8, 9 an 10), there is a more clever way of doing this, due to P. Montgomery which can save 10 to 20 percent of the running time, and this is not a negligible saving since it is an absolutely basic operation. We refer to his paper [Mon1] for the description of this method.

1.1.3 Base Fields and Rings

Many of the algorithms that we give (for example the linear algebra algorithms of Chapter 2 or some of the algorithms for working with polynomials in Chapter 3) are valid over any base ring or field R where we know how to compute. We must emphasize however that the behavior of these algorithms will be quite different depending on the base ring. Let us look at the most important examples.

The simplest rings are the rings $R = \mathbb{Z}/N\mathbb{Z}$, especially when N is small. Operations in R are simply operations "modulo N" and the elements of R can always be represented by an integer less than N, hence of bounded size. Using the standard algorithms mentioned in the preceding section, and a suitable version of Euclid's extended algorithm to perform division (see Section 1.3.2), all operations need only $O(\ln^2 N)$ bit operations (in fact $O(1)$ since N is considered as fixed!). An important special case of these rings R is when $N = p$ is a prime, and then $R = \mathbb{F}_p$ the finite field with p elements. More generally, it is easy to see that operations on any finite field $\mathbb{F}_q$ with $q = p^k$ can be done quickly.

The next example is that of $R = \mathbb{Z}$. In many algorithms, it is possible to give an upper bound N on the size of the numbers to be handled. In this case we are back in the preceding situation, except that the bound N is no longer fixed, hence the running time of the basic operations is really $O(\ln^2 N)$ bit operations and not $O(1)$. Unfortunately, in most algorithms some divisions are needed, hence we are no longer working in $\mathbb{Z}$ but rather in $\mathbb{Q}$. It is possible to rewrite some of these algorithms so that non-integral rational numbers never occur (see for example the Gauss-Bareiss Algorithm 2.2.6, the integral LLL Algorithm 2.6.7, the sub-resultant Algorithms 3.3.1 and 3.3.7). These versions are then preferable.

The third example is when $R = \mathbb{Q}$. The main phenomenon which occurs in practically all algorithms here is "coefficient explosion". This means that in the course of the algorithm the numerator and denominators of the rational numbers which occur become very large; their size is almost impossible to control. The main reason for this is that the numerator and denominator of the sum or difference of two rational numbers is usually of the same order of magnitude as those of their product. Consequently it is not easy to give running times in bit operations for algorithms using rational numbers.

The fourth example is that of $R = \mathbb{R}$ (or $R = \mathbb{C}$). A new phenomenon occurs here. How can we represent a real number? The truthful answer is that it is in practice impossible, not only because the set $\mathbb{R}$ is uncountable, but also because it will always be impossible for an algorithm to tell whether two real numbers are equal, since this requires in general an infinite amount of time (on the other hand if two real numbers are different, it is possible to prove it by computing them to sufficient accuracy). So we must be content with approximations (or with interval arithmetic, i.e. we give for each real number involved in an algorithm a rational lower and upper bound), increasing the closeness of the approximation to suit our needs. A nasty specter is waiting for us in the dark, which has haunted generations of numerical analysts: numerical instability. We will see an example of this in the case of the LLL algorithm (see Remark (4) after Algorithm 2.6.3). Since this is not a book on numerical analysis, we do not dwell on this problem, but it should be kept in mind.

As far as the bit complexity of the basic operations are concerned, since we must work with limited accuracy the situation is analogous to that of $\mathbb{Z}$ when an upper bound N is known. If the accuracy used for the real number is of the order of $1/N$, the number of bit operations for performing the basic operations is $O(\ln^2 N)$.

Although not much used in this book, a last example I would like to mention is that of $R = \mathbb{Q}_p$, the field of p-adic numbers. This is similar to the case of real numbers in that we must work with a limited precision, hence the running times are of the same order of magnitude. Since the p-adic valuation is non-Archimedean, i.e. the accuracy of the sum or product of p-adic numbers with a given accuracy is at least of the same accuracy, the phenomenon of numerical instability essentially disappears.

1.1.4 Notations

We will use Knuth's notations, which have become a *de facto* standard in the theory of algorithms. Also, some algorithms are directly adapted from Knuth (why change a well written algorithm?). However the algorithmic style of writing used by Knuth is not well suited to structured programming. The reader may therefore find it completely straightforward to write the corresponding programs in assembly language, Basic or Fortran, say, but may find it slightly less so to write them in Pascal or in C.

A warning: presenting an algorithms as a series of steps as is done in this book is only one of the ways in which an algorithm can be described. The presentation may look old-fashioned to some readers, but in the author's opinion it is the best way to explain all the details of an algorithm. In particular it is perhaps better than using some pseudo-Pascal language (pseudo-code). Of course, this is debatable, but this is the choice that has been made in this book. Note however that, as a consequence, the reader should read as carefully as possible the exact phrasing of the algorithm, as well as the accompanying explanations, to avoid any possible ambiguity. This is particularly true in `if`

(conditional) expressions. Some additional explanation is sometimes added to diminish the possibility of ambiguity. For example, if the `if` condition is not satisfied, the usual word used is **otherwise**. If `if` expressions are nested, one of them will use **otherwise**, and the other will usually use **else**. I admit that this is not a very elegant solution.

A typical example is step 7 in Algorithm 6.2.9. The initial statement `If` $c = 0$ **do the following:** implies that the whole step will be executed only if $c = 0$, and must be skipped if $c \neq 0$. Then there is the expression `if` $j = i$ followed by an **otherwise**, and nested inside the **otherwise clause** is another `if` $\dim(...) < n$, and the **else go to step** 7 which follows refers to this last `if`, i.e. we go to step 7 if $\dim(...) \geq n$.

I apologize to the reader if this causes any confusion, but I believe that this style of presentation is a good compromise.

$\lfloor x \rfloor$ denotes the floor of x, i.e. the largest integer less than or equal to x. Thus $\lfloor 3.4 \rfloor = 3$, $\lfloor -3.4 \rfloor = -4$.

$\lceil x \rceil$ denotes the ceiling of x, i.e. the smallest integer greater than or equal to x. We have $\lceil x \rceil = -\lfloor -x \rfloor$.

$\lfloor x \rceil$ denotes an integer nearest to x, i.e. $\lfloor x \rceil = \lfloor x + 1/2 \rfloor$.

$[a, b[$ denotes the real interval from a to b including a but excluding b. Similarly $]a, b]$ includes b and excludes a, and $]a, b[$ is the open interval excluding a and b. (This differs from the American notations $[a, b)$, $(a, b]$ and (a, b) which in my opinion are terrible. In particular, in this book (a, b) will usually mean the GCD of a and b, and sometimes the ordered pair (a, b).)

$\lg x$ denotes the base 2 logarithm of x.

If E is a finite set, $|E|$ denotes the cardinality of E.

If A is a matrix, A^t denotes the transpose of the matrix A. A $1 \times n$ (resp. $n \times 1$) matrix is called a row (resp. column) vector. The reader is warned that many authors use a different notation where the transpose sign is put on the left of the matrix.

If a and b are integers with $b \neq 0$, then except when explicitly mentioned otherwise, $a \bmod b$ denotes the *non-negative* remainder in the Euclidean division of a by b, i.e. the unique number r such that $a \equiv r \pmod{b}$ and $0 \leq r < |b|$.

The notation $d \mid n$ means that d divides n, while $d \| n$ will mean that $d \mid n$ and $(d, n/d) = 1$. Furthermore, the notations $p \mid n$ and $p^\alpha \| n$ are always taken to imply that p is prime, so for example $p^\alpha \| n$ means that p^α is the highest power of p dividing n.

Finally, if a and b are elements in a Euclidean ring (typically $\mathbb{Z}$ or the ring of polynomials over a field), we will denote the greatest common divisor (abbreviated GCD in the text) of a and b by $\gcd(a, b)$, or simply by (a, b) when there is no risk of confusion.

1.2 The Powering Algorithms

In almost every non-trivial algorithm in number theory, it is necessary at some point to compute the n-th power of an element in a group, where n may be some very large integer (i.e. for instance greater than 10^{100}). That this is actually possible and very easy is fundamental and one of the first things that one must understand in algorithmic number theory. These algorithms are general and can be used in any group. In fact, when the exponent is non-negative, they can be used in any monoid with unit. We give an abstract version, which can be trivially adapted for any specific situation.

Let $(G, \times)$ be a group. We want to compute g^n for $g \in G$ and $n \in \mathbb{Z}$ in an efficient manner. Assume for example that $n > 0$. The naïve method requires $n-1$ group multiplications. We can however do much better (A note: although Gauss was very proficient in hand calculations, he seems to have missed this method.) The idea is as follows. If $n = \sum_i \epsilon_i 2^i$ is the base 2 expansion of n with $\epsilon_i = 0$ or 1, then

$$g^n = \prod_{\epsilon_i = 1} \left(g^{2^i}\right),$$

hence if we keep track in an auxiliary variable of the quantities g^{2^i} which we compute by successive squarings, we obtain the following algorithm.

Algorithm 1.2.1 (Right-Left Binary). Given $g \in G$ and $n \in \mathbb{Z}$, this algorithm computes g^n in G. We write 1 for the unit element of G.

1. [Initialize] Set $y \leftarrow 1$. If $n = 0$, output y and terminate. If $n < 0$ let $N \leftarrow -n$ and $z \leftarrow g^{-1}$. Otherwise, set $N \leftarrow n$ and $z \leftarrow g$.
2. [Multiply?] If N is odd set $y \leftarrow z \cdot y$.
3. [Halve N] Set $N \leftarrow \lfloor N/2 \rfloor$. If $N = 0$, output y as the answer and terminate the algorithm. Otherwise, set $z \leftarrow z \cdot z$ and go to step 2.

Examining this algorithm shows that the number of multiplication steps is equal to the number of binary digits of $|n|$ plus the number of ones in the binary representation of $|n|$ minus 1. So, it is at most equal to $2\lfloor \lg |n| \rfloor + 1$, and on average approximately equal to $1.5 \lg |n|$. Hence, if one can compute rapidly in G, it is not unreasonable to have exponents with several million decimal digits. For example, if $G = (\mathbb{Z}/m\mathbb{Z})^*$, the time of the powering algorithm is $O(\ln^2 m \ln |n|)$, since one multiplication in G takes time $O(\ln^2 m)$.

The validity of Algorithm 1.2.1 can be checked immediately by noticing that at the start of step 2 one has $g^n = y \cdot z^N$. This corresponds to a right-to-left scan of the binary digits of $|n|$.

We can make several changes to this basic algorithm. First, we can write a similar algorithm based on a left to right scan of the binary digits of $|n|$. In other words, we use the formula $g^n = (g^{n/2})^2$ if n is even and $g^n = g \cdot (g^{(n-1)/2})^2$ if n is odd.

This assumes however that we know the position of the leftmost bit of $|n|$ (or that we have taken the time to look for it beforehand), i.e. that we know the integer e such that $2^e \leq |n| < 2^{e+1}$. Such an integer can be found using a standard binary search on the binary digits of n, hence the time taken to find it is $O(\lg \lg |n|)$, and this is completely negligible with respect to the other operations. This leads to the following algorithm.

Algorithm 1.2.2 (Left-Right Binary). Given $g \in G$ and $n \in \mathbb{Z}$, this algorithm computes g^n in G. If $n \neq 0$, we assume also given the unique integer e such that $2^e \leq |n| < 2^{e+1}$. We write 1 for the unit element of G.

1. [Initialize] If $n = 0$, output 1 and terminate. If $n < 0$ set $N \leftarrow -n$ and $z \leftarrow g^{-1}$. Otherwise, set $N \leftarrow n$ and $z \leftarrow g$. Finally, set $y \leftarrow z$, $E \leftarrow 2^e$, $N \leftarrow N - E$.
2. [Finished?] If $E = 1$, output y and terminate the algorithm. Otherwise, set $E \leftarrow E/2$.
3. [Multiply?] Set $y \leftarrow y \cdot y$ and if $N \geq E$, set $N \leftarrow N - E$ and $y \leftarrow y \cdot z$. Go to step 2.

Note that E takes as values the decreasing powers of 2 from 2^e down to 1, hence when implementing this algorithm, all operations using E must be thought of as bit operations. For example, instead of keeping explicitly the (large) number E, one can just keep its exponent (which will go from e down to 0). Similarly, one does not really subtract E from N or compare N with E, but simply look whether a particular bit of N is 0 or not. To be specific, assume that we have written a little program $\text{bit}(N, f)$ which outputs bit number f of N, bit 0 being, by definition, the least significant bit. Then we can rewrite Algorithm 1.2.2 as follows.

Algorithm 1.2.3 (Left-Right Binary, Using Bits). Given $g \in G$ and $n \in \mathbb{Z}$, this algorithm computes g^n in G. If $n \neq 0$, we assume also that we are given the unique integer e such that $2^e \leq |n| < 2^{e+1}$. We write 1 for the unit element of G.

1. [Initialize] If $n = 0$, output 1 and terminate. If $n < 0$ set $N \leftarrow -n$ and $z \leftarrow g^{-1}$. Otherwise, set $N \leftarrow n$ and $z \leftarrow g$. Finally, set $y \leftarrow z$, $f \leftarrow e$.
2. [Finished?] If $f = 0$, output y and terminate the algorithm. Otherwise, set $f \leftarrow f - 1$.
3. [Multiply?] Set $y \leftarrow y \cdot y$ and if $\text{bit}(N, f) = 1$, set $y \leftarrow y \cdot z$. Go to step 2.

The main advantage of this algorithm over Algorithm 1.2.1 is that in step 3 above, z is always the initial g (or its inverse if $n < 0$). Hence, if g is represented by a small integer, this may mean a linear time multiplication instead of a quadratic time one. For example, if $G = (\mathbb{Z}/m\mathbb{Z})^*$ and if g (or g^{-1} if $n < 0$) is represented by the class of a single precision integer, the

running time of Algorithms 1.2.2 and 1.2.3 will be in average up to 1.5 times faster than Algorithm 1.2.1.

Algorithm 1.2.3 can be improved by making use of the representation of $|n|$ in a base equal to a power of 2, instead of base 2 itself. In this case, only the left-right version exists.

This is done as follows (we may assume $n > 0$). Choose a suitable positive integer k (we will see in the analysis how to choose it optimally). Precompute g^2 and by induction the odd powers g^3, g^5, ..., g^{2^k-1}, and initialize y to g as in Algorithm 1.2.3. Now if we scan the 2^k-representation of $|n|$ from left to right (i.e. k bits at a time of the binary representation), we will encounter digits a in base 2^k, hence such that $0 \le a < 2^k$. If $a = 0$, we square k times our current y. If $a \neq 0$, we can write $a = 2^t b$ with b odd and less than 2^k, and $0 \le t < k$. We must set $y \leftarrow y^{2^k} \cdot g^{2^t b}$, and this is done by computing first $y^{2^{k-t}} \cdot g^b$ (which involves $k-t$ squarings plus one multiplication since g^b has been precomputed), then squaring t times the result. This leads to the following algorithm. Here we assume that we have an algorithm digit(k, N, f) which gives digit number f of N expressed in base 2^k.

Algorithm 1.2.4 (Left-Right Base 2^k). Given $g \in G$ and $n \in \mathbb{Z}$, this algorithm computes g^n in G. If $n \neq 0$, we assume also given the unique integer e such that $2^{ke} \le |n| < 2^{k(e+1)}$. We write 1 for the unit element of G.

1. [Initialize] If $n = 0$, output 1 and terminate. If $n < 0$ set $N \leftarrow -n$ and $z \leftarrow g^{-1}$. Otherwise, set $N \leftarrow n$ and $z \leftarrow g$. Finally set $f \leftarrow e$.
2. [Precomputations] Compute and store z^3, z^5, ..., z^{2^k-1}.
3. [Multiply] Set $a \leftarrow$ digit(k, N, f). If $a = 0$, repeat k times $y \leftarrow y \cdot y$. Otherwise, write $a = 2^t b$ with b odd, and if $f \neq e$ repeat $k - t$ times $y \leftarrow y \cdot y$ and set $y \leftarrow y \cdot z^b$, while if $f = e$ set $y \leftarrow z^b$ (using the precomputed value of z^b), and finally (still if $a \neq 0$) repeat t times $y \leftarrow y \cdot y$.
4. [Finished?] If $f = 0$, output y and terminate the algorithm. Otherwise, set $f \leftarrow f - 1$ and go to step 3.

Implementation Remark. Although the splitting of a in the form $2^t b$ takes very little time compared to the rest of the algorithm, it is a nuisance to have to repeat it all the time. Hence, we suggest precomputing all pairs (t, b) for a given k (including $(k, 0)$ for $a = 0$) so that t and b can be found simply by table lookup. Note that this precomputation depends only on the value of k chosen for Algorithm 1.2.4, and not on the actual value of the exponent n.

Let us now analyze the average behavior of Algorithm 1.2.4 so that we can choose k optimally. As we have already explained, we will regard as negligible the time spent in computing e or in extracting bits or digits in base 2^k.

The precomputations require 2^{k-1} multiplications. The total number of squarings is exactly the same as in the binary algorithm, i.e. $\lfloor \lg |n| \rfloor$, and the number of multiplications is equal to the number of non-zero digits of $|n|$ in base 2^k, i.e. on average

$$\left(1-\frac{1}{2^k}\right)\left(\left\lfloor\frac{\lg|n|}{k}\right\rfloor+1\right),$$

so the total number of multiplications which are not squarings is on average approximately equal to

$$m(k)=2^{k-1}+\left(\frac{2^k-1}{k2^k}\right)\lg|n|.$$

Now, if we compute $m(k+1)-m(k)$, we see that it is non-negative as long as

$$\lg|n|\leq\frac{k(k+1)2^{2k}}{2^{k+1}-k-2}.$$

Hence, for the highest efficiency, one should choose k equal to the smallest integer satisfying the above inequality, and this gives $k=1$ for $|n|\leq 256$, $k=2$ for $|n|\leq 2^{24}$, etc .... For example, if $|n|$ has between 60 and 162 decimal digits, the optimal value of k is $k=5$. For a more specific example, assume that n has 100 decimal digits (i.e. $\lg n$ approximately equal to 332) and that the time for squaring is about 3/4 of the time for multiplication (this is quite a reasonable assumption). Then, counting multiplication steps, the ordinary binary algorithm takes on average $(3/4)332+332/2=415$ steps. On the other hand, the base 2^5 algorithm takes on average $(3/4)332+16+(31/160)332\approx 329$ multiplication steps, an improvement of more than 20%.

There is however another point to take into account. When, for instance $G=(\mathbb{Z}/m\mathbb{Z})^*$ and g (or g^{-1} when $n<0$) is represented by the (residue) class of a single precision integer, replacing multiplication by g by multiplication by its small odd powers may have the disadvantage compared to Algorithm 1.2.3 that these powers may not be single precision. Hence, in this case, it may be preferable, either to use Algorithm 1.2.3, or to use the highest power of k less than or equal to the optimal one which keeps all the z^b with b odd and $1\leq b\leq 2^k-1$ represented by single precision integers.

(A long text should be inserted here, but no place to do this (see page 45).)

Quite a different way to improve on Algorithm 1.2.1 is to try to find a near optimal "addition chain" for $|n|$, and this also can lead to improvements, especially when the same exponent is used repeatedly (see [BCS]. For a detailed discussion of addition chains, see [Knu2].) In practice, we suggest using the flexible 2^k-algorithm for a suitable value of k.

The powering algorithm is used very often with the ring $\mathbb{Z}/m\mathbb{Z}$. In this case multiplication does not give a group law, but the algorithm is valid nonetheless if either n is non-negative or if g is an invertible element. Furthermore, the group multiplication is "multiplication followed by reduction modulo m". Depending on the size of m, it may be worthwhile to not do the reductions each time, but to do them only when necessary to avoid overflow or loss of time.

We will use the powering algorithm in many other contexts in this book, in particular when computing in class groups of number fields, or when working with elliptic curves over finite fields.

Note that for many groups it is possible (and desirable) to write a squaring routine which is faster than the general-purpose multiplication routine. In situations where the powering algorithm is used intensively, it is essential to use this squaring routine when multiplications of the type $y \leftarrow y \cdot y$ are encountered.

1.3 Euclid's Algorithms

We now consider the problem of computing the GCD of two integers a and b. The naïve answer to this problem would be to factor a and b, and then multiply together the common prime factors raised to suitable powers. Indeed, this method works well when a and b are *very* small, say less than 100, or when a or b is known to be prime (then a single division is sufficient). In general this is not feasible, because one of the important facts of life in number theory is that factorization is difficult and slow. We will have many occasions to come back to this. Hence, we must use better methods to compute GCD's. This is done using Euclid's algorithm, probably the oldest and most important algorithm in number theory.

Although very simple, this algorithm has several variants, and, because of its usefulness, we are going to study it in detail. We shall write (a, b) for the GCD of a and b when there is no risk of confusion with the pair (a, b). By definition, (a, b) is the unique non-negative generator of the additive subgroup of $\mathbb{Z}$ generated by a and b. In particular, $(a, 0) = (0, a) = |a|$ and $(a, b) = (|a|, |b|)$. Hence we can always assume that a and b are non-negative.

1.3.1 Euclid's and Lehmer's Algorithms

Euclid's algorithm is as follows:

Algorithm 1.3.1 (Euclid). Given two non-negative integers a and b, this algorithm finds their GCD.

1. [Finished?] If $b = 0$ then output a as the answer and terminate the algorithm.
2. [Euclidean step] Set $r \leftarrow a \bmod b$, $a \leftarrow b$, $b \leftarrow r$ and go to step 1.

If either a or b is less than a given number N, the number of Euclidean steps in this algorithm is bounded by a constant times $\ln N$, in both the worst case and on average. More precisely we have the following theorem (see [Knu2]):

Theorem 1.3.2. *Assume that a and b are randomly distributed between 1 and N. Then*

(1) *The number of Euclidean steps is at most equal to*

$$\left\lceil \frac{\ln(\sqrt{5}N)}{\ln((1+\sqrt{5})/2)} \right\rceil - 2 \approx 2.078 \ln N + 1.672.$$

(2) *The average number of Euclidean steps is approximately equal to*

$$\frac{12 \ln 2}{\pi^2} \ln N + 0.14 \approx 0.843 \ln N + 0.14.$$

However, Algorithm 1.3.1 is far from being the whole story. First, it is not well suited to handling large numbers (in our sense, say numbers with 50 or 100 decimal digits). This is because each Euclidean step requires a long division, which takes time $O(\ln^2 N)$. When carelessly programmed, the algorithm takes time $O(\ln^3 N)$. If, however, at each step the precision is decreased as a function of a and b, and if one also notices that the time to compute a Euclidean step $a = bq + r$ is $O((\ln a)(\ln q + 1))$, then the total time is bounded by $O((\ln N)((\sum \ln q) + O(\ln N)))$. But $\sum \ln q = \ln \prod q \leq \ln a \leq \ln N$, hence if programmed carefully, the running time is only $O(\ln^2 N)$. There is a useful variant due to Lehmer which also brings down the running time to $O(\ln^2 N)$. The idea is that the Euclidean quotient depends generally only on the first few digits of the numbers. Therefore it can usually be obtained using a single precision calculation. The following algorithm is taken directly from Knuth. Let $M = m^p$ be the base used for multi-precision numbers. Typical choices are $m = 2$, $p = 15, 16, 31$, or 32, or $m = 10$, $p = 4$ or 9.

Algorithm 1.3.3 (Lehmer). Let a and b be non-negative multi-precision integers, and assume that $a \geq b$. This algorithm computes (a, b), using the following auxiliary variables. $\hat{a}$, $\hat{b}$, A, B, C, D, T and q are single precision (i.e. less than M), and t and r are multi-precision variables.

1. [Initialize] If $b < M$, i.e. is single precision, compute (a, b) using Algorithm 1.3.1 and terminate. Otherwise, let $\hat{a}$ (resp. $\hat{b}$) be the single precision number formed by the highest non-zero base M digit of a (resp. b). Set $A \leftarrow 1$, $B \leftarrow 0$, $C \leftarrow 0$, $D \leftarrow 1$.

2. [Test quotient] If $\hat{b} + C = 0$ or $\hat{b} + D = 0$ go to step 4. Otherwise, set $q \leftarrow \lfloor (\hat{a} + A)/(\hat{b} + C) \rfloor$. If $q \neq \lfloor (\hat{a} + B)/(\hat{b} + D) \rfloor$, go to step 4. Note that one always has the conditions

$$0 \leq \hat{a} + A \leq M, \quad 0 \leq \hat{b} + C < M,$$

$$0 \leq \hat{a} + B < M, \quad 0 \leq \hat{b} + D \leq M.$$

Notice that one can have a single precision overflow in this step, which must be taken into account. (This can occur only if $\hat{a} = M - 1$ and $A = 1$ or if $\hat{b} = M - 1$ and $D = 1$.)

3. [Euclidean step] Set $T \leftarrow A - qC$, $A \leftarrow C$, $C \leftarrow T$, $T \leftarrow B - qD$, $B \leftarrow D$, $D \leftarrow T$, $T \leftarrow \hat{a} - q\hat{b}$, $\hat{a} \leftarrow \hat{b}$, $\hat{b} \leftarrow T$ and go to step 2 (all these operations are single precision operations).
4. [Multi-precision step] If $B = 0$, set $t \leftarrow a \bmod b$, $a \leftarrow b$, $b \leftarrow t$, using multi-precision division (this happens with a very small probability, on the order of $1.4/M$) and go to step 1. Otherwise, set $t \leftarrow Aa$, $t \leftarrow t + Bb$, $r \leftarrow Ca$, $r \leftarrow r + Db$, $a \leftarrow t$, $b \leftarrow r$, using linear-time multi-precision operations, and go to step 1.

Note that the number of steps in this algorithm will be the same as in Algorithm 1.3.1, i.e. $O(\ln N)$ if a and b are less than N, but each loop now consists only of linear time operations (except for the case $B = 0$ in step 4 which is so rare as not to matter in practice). Therefore, even without using variable precision, the running time is now only of order $O(\ln^2 N)$ and not $O(\ln^3 N)$. Of course, there is much more bookkeeping involved, so it is not clear how large N must be before a particular implementation of this algorithm becomes faster than a crude implementation of Algorithm 1.3.1. Or, even whether a careful implementation of Algorithm 1.3.1 will not compete favorably in practice. Testing needs to be done before choosing which of these algorithms to use.

Another variant of Euclid's algorithm which is also useful in practice is the so-called binary algorithm. Here, no long division steps are used, except at the beginning, instead only subtraction steps and divisions by 2, which are simply integer shifts. The number of steps needed is greater, but the operations used are much faster, and so there is a net gain, which can be quite large for multi-precision numbers. Furthermore, using subtractions instead of divisions is quite reasonable in any case, since most Euclidean quotients are small. More precisely, we can state:

Theorem 1.3.4. *In a suitable sense, the probability $P(q)$ that a Euclidean quotient be equal to q is*

$$P(q) = \lg((q+1)^2/((q+1)^2 - 1)).$$

For example, $P(1) = 0.41504\ldots$, $P(2) = 0.16992\ldots$, $P(3) = 0.09311\ldots$, $P(4) = 0.05890\ldots$.

For example, from this theorem, one can see that the probability of occurrence of $B = 0$ in step 4 of Algorithm 1.3.3 is $\lg(1 + 1/M)$, and this is negligible in practice.

One version of the binary algorithm is as follows.

Algorithm 1.3.5 (Binary GCD). Given two non-negative integers a and b, this algorithm finds their GCD.

1. [Reduce size once] If $a < b$ exchange a and b. Now if $b = 0$, output a and terminate the algorithm. Otherwise, set $r \leftarrow a \bmod b$, $a \leftarrow b$ and $b \leftarrow r$.

2. [Compute power of 2] If $b = 0$ output a and terminate the algorithm. Otherwise, set $k \leftarrow 0$, and then while a and b are both even, set $k \leftarrow k + 1$, $a \leftarrow a/2$, $b \leftarrow b/2$.

3. [Remove initial powers of 2] If a is even, repeat $a \leftarrow a/2$ until a is odd. Otherwise, if b is even, repeat $b \leftarrow b/2$ until b is odd.

4. [Subtract] (Here a and b are both odd.) Set $t \leftarrow (a - b)/2$. If $t = 0$, output $2^k a$ and terminate the algorithm.

5. [Loop] While t is even, set $t \leftarrow t/2$. Then if $t > 0$ set $a \leftarrow t$, else set $b \leftarrow -t$ and go to step 4.

Remarks.

(1) The binary algorithm is especially well suited for computing the GCD of multi-precision numbers. This is because no divisions are performed, except on the first step. Hence we suggest using it systematically in this case.

(2) All the divisions by 2 performed in this algorithm must be done using shifts or Boolean operations, otherwise the algorithm loses much of its attractiveness. In particular, it may be worthwhile to program it in a low-level language, and even in assembly language, if it is going to be used extensively. Note that some applications, such as computing in class groups, use GCD as a basic operation, hence it is essential to optimize the speed of the algorithm for these applications.

(3) One could directly start the binary algorithm in step 2, avoiding division altogether. We feel however that this is not such a good idea, since a and b may have widely differing magnitudes, and step 1 ensures that we will work on numbers at most the size of the *smallest* of the two numbers a and b, and not of the largest, as would be the case if we avoided step 1. In addition, it is quite common for b to divide a when starting the algorithm. In this case, of course, the algorithm immediately terminates after step 1.

(4) Note that the sign of t in step 4 of the algorithm enables the algorithm to keep track of the larger of a and b, so that we can replace the larger of the two by $|t|$ in step 5. We can also keep track of this data in a separate variable and thereby work only with non-negative numbers.

(5) Finally, note that the binary algorithm can use the ideas of Algorithm 1.3.3 for multi-precision numbers. The resulting algorithm is complex and its efficiency is implementation dependent. For more details, see [Knu2 p.599].

The proof of the validity of the binary algorithm is easy and left to the reader. On the other hand, a detailed analysis of the average running time of the binary algorithm is a challenging mathematical problem (see [Knu2] once again). Evidently, as was the case for Euclid's algorithm, the running time will be $O(\ln^2 N)$ bit operations when suitably implemented, where N is an upper bound on the size of the inputs a and b. The mathematical problem is to find

an asymptotic estimate for the number of steps and the number of shifts performed in Algorithm 1.3.5, but this has an influence only on the O constant, not on the qualitative behavior. □

1.3.2 Euclid's Extended Algorithms

The information given by Euclid's algorithm is not always sufficient for many problems. In particular, by definition of the GCD, if $d = (a, b)$ there exists integers u and v such that $au + bv = d$. It is often necessary to extend Euclid's algorithm so as to be able to compute u and v. While u and v are not unique, u is defined modulo b/d, and v is defined modulo a/d.

There are two ways of doing this. One is by storing the Euclidean quotients as they come along, and then, once d is found, backtracking to the initial values. This method is the most efficient, but can require a lot of storage. In some situations where this information is used extensively (such as Shanks's and Atkin's NUCOMP in Section 5.4.2), any little gain should be taken, and so one should do it this way.

The other method requires very little storage and is only slightly slower. This requires using a few auxiliary variables so as to do the computations as we go along. We first give a version which does not take into account multi-precision numbers.

Algorithm 1.3.6 (Euclid Extended). Given non-negative integers a and b, this algorithm determines (u, v, d) such that $au + bv = d$ and $d = (a, b)$. We use auxiliary variables v_1, v_3, t_1, t_3.

1. [Initialize] Set $u \leftarrow 1$, $d \leftarrow a$. If $b = 0$, set $v \leftarrow 0$ and terminate the algorithm, otherwise set $v_1 \leftarrow 0$ and $v_3 \leftarrow b$.
2. [Finished?] If $v_3 = 0$ then set $v \leftarrow (d - au)/b$ and terminate the algorithm.
3. [Euclidean step] Let $q \leftarrow \lfloor d/v_3 \rfloor$ and simultaneously $t_3 \leftarrow d \bmod v_3$. Then set $t_1 \leftarrow u - qv_1$, $u \leftarrow v_1$, $d \leftarrow v_3$, $v_1 \leftarrow t_1$, $v_3 \leftarrow t_3$ and go to step 2.

"Simultaneously" in step 3 means that if this algorithm is implemented in assembly language, then, since the division instruction usually gives both the quotient and remainder, this should of course be used. Even if this algorithm is not programmed in assembly language, but a and b are multi-precision numbers, the division routine in the multi-precision library should also return both quotient and remainder. Note also that in step 2, the division of $d - au$ by b is exact.

Proof of the Algorithm. Introduce three more variables v_2, t_2 and v. We want the following relations to hold each time one begins step 2:

$$at_1 + bt_2 = t_3, \quad au + bv = d, \quad av_1 + bv_2 = v_3.$$

For this to be true after the initialization step, it suffices to set $v \leftarrow 0$, $v_2 \leftarrow 1$. (It is not necessary to initialize the t variables.) Then, it is easy to check that step 3 preserves these relations if we update suitably the three auxiliary variables (by $(v_2, t_2, v) \leftarrow (t_2, v - qv_2, v_2)$). Therefore, at the end of the algorithm, d contains the GCD (since we have simply added some extra work to the initial Euclidean algorithm), and we also have $au + bv = d$. □

As an exercise, the reader can show that at the end of the algorithm, we have $v_1 = \pm b/d$ (and $v_2 = \mp a/d$ in the proof), and that throughout the algorithm, $|v_1|$, $|u|$, $|t_1|$ stay less than or equal to b/d (and $|v_2|$, $|v|$, $|t_2|$ stay less than or equal to a/d).

This algorithm can be improved for multi-precision numbers exactly as in Lehmer's Algorithm 1.3.3. Since it is a simple blend of Algorithms 1.3.3 and 1.3.5, we do not give a detailed proof. (Notice however that the variables d and v_3 have become a and b.)

Algorithm 1.3.7 (Lehmer Extended). Let a and b be non-negative multi-precision integers, and assume that $a \geq b$. This algorithm computes (u, v, d) such that $au + bv = d = (a, b)$, using the following auxiliary variables. $\hat{a}$, $\hat{b}$, A, B, C, D, T and q are single precision (i.e. less than M), and t, r, v_1, v_3 are multi-precision variables.

1. [Initialize] Set $u \leftarrow 1$, $v_1 \leftarrow 0$.
2. [Finished?] If $b < M$, i.e. is single precision, compute (u, v, d) using Algorithm 1.3.6 and terminate. Otherwise, let $\hat{a}$ (resp. $\hat{b}$) be the single precision number formed by the p most significant digits of a (resp. b). Set $A \leftarrow 1$, $B \leftarrow 0$, $C \leftarrow 0$, $D \leftarrow 1$.
3. [Test quotient] If $\hat{b} + C = 0$ or $\hat{b} + D = 0$ go to step 5. Otherwise, set $q \leftarrow \lfloor (\hat{a} + A)/(\hat{b} + C) \rfloor$. If $q \neq \lfloor (\hat{a} + B)/(\hat{b} + D) \rfloor$, go to step 5.
4. [Euclidean step] Set $T \leftarrow A - qC$, $A \leftarrow C$, $C \leftarrow T$, $T \leftarrow B - qD$, $B \leftarrow D$, $D \leftarrow T$, $T \leftarrow \hat{a} - q\hat{b}$, $\hat{a} \leftarrow \hat{b}$, $\hat{b} \leftarrow T$ and go to step 3 (all these operations are single precision operations).
5. [Multi-precision step] If $B = 0$, set $q \leftarrow \lfloor a/b \rfloor$ and simultaneously $t \leftarrow a \bmod b$ using multi-precision division, then $a \leftarrow b$, $b \leftarrow t$, $t \leftarrow u - qv_1$, $u \leftarrow v_1$, $v_1 \leftarrow t$ and go to step 2.

   Otherwise, set $t \leftarrow Aa$, $t \leftarrow t + Bb$, $r \leftarrow Ca$, $r \leftarrow r + Db$, $a \leftarrow t$, $b \leftarrow r$, $t \leftarrow Au$, $t \leftarrow t + Bv_1$, $r \leftarrow Cu$, $r \leftarrow r + Dv_1$, $u \leftarrow t$, $v_1 \leftarrow r$ using linear-time multi-precision operations, and go to step 2.

In a similar way, the binary algorithm can be extended to find u and v. The algorithm is as follows.

Algorithm 1.3.8 (Binary Extended). Given non-negative integers a and b, this algorithm determines (u, v, d) such that $au + bv = d$ and $d = (a, b)$. We use auxiliary variables v_1, v_3, t_1, t_3, and two Boolean flags f_1 and f_2.

1. [Reduce size once] If $a < b$ exchange a and b and set $f_1 \leftarrow 1$, otherwise set $f_1 \leftarrow 0$. Now if $b = 0$, output $(1, 0, a)$ if $f_1 = 0$, $(0, 1, a)$ if $f_1 = 1$ and terminate the algorithm. Otherwise, let $a = bq + r$ be the Euclidean division of a by b, where $0 \leq r < b$, and set $a \leftarrow b$ and $b \leftarrow r$.
2. [Compute power of 2] If $b = 0$, output $(0, 1, a)$ if $f_1 = 0$, $(1, 0, a)$ if $f_1 = 1$ and terminate the algorithm. Otherwise, set $k \leftarrow 0$, and while a and b are both even, set $k \leftarrow k + 1$, $a \leftarrow a/2$, $b \leftarrow b/2$.
3. [Initialize] If b is even, exchange a and b and set $f_2 \leftarrow 1$, otherwise set $f_2 \leftarrow 0$. Then set $u \leftarrow 1$, $d \leftarrow a$, $v_1 \leftarrow b$, $v_3 \leftarrow b$. If a is odd, set $t_1 \leftarrow 0$, $t_3 \leftarrow -b$ and go to step 5, else set $t_1 \leftarrow (1 + b)/2$, $t_3 \leftarrow a/2$.
4. [Remove powers of 2] If t_3 is even do as follows. Set $t_3 \leftarrow t_3/2$, $t_1 \leftarrow t_1/2$ if t_1 is even and $t_1 \leftarrow (t_1 + b)/2$ if t_1 is odd, and repeat step 4.
5. [Loop] If $t_3 > 0$, set $u \leftarrow t_1$ and $d \leftarrow t_3$, otherwise, set $v_1 \leftarrow b - t_1$, $v_3 \leftarrow -t_3$.
6. [Subtract] Set $t_1 \leftarrow u - v_1$, $t_3 \leftarrow d - v_3$. If $t_1 < 0$, set $t_1 \leftarrow t_1 + b$. Finally, if $t_3 \neq 0$, go to step 4.
7. [Terminate] Set $v \leftarrow (d - au)/b$ and $d \leftarrow 2^k d$. If $f_2 = 1$ exchange u and v. Then set $u \leftarrow u - vq$. Finally, output (u, v, d) if $f_1 = 1$, (v, u, d) if $f_1 = 0$, and terminate the algorithm.

Proof. The proof is similar to that of Algorithm 1.3.6. We introduce three more variables v_2, t_2 and v and we require that at the start of step 4 we always have

$$At_1 + Bt_2 = t_3\,, \quad Au + Bv = d, \quad Av_1 + Bv_2 = v_3,$$

where A and B are the values of a and b after step 3. For this to be true, we must initialize them by setting (in step 3) $v \leftarrow 0$, $v_2 \leftarrow 1 - a$ and $t_2 \leftarrow -1$ if a is odd, $t_2 \leftarrow -a/2$ if a is even. After this, the three relations will continue to be true provided we suitably update v_2, t_2 and v. Since, when the algorithm terminates d will be the GCD of A and B, it suffices to backtrack from both the division step and the exchanges done in the first few steps in order to obtain the correct values of u and v (as is done in step 7). We leave the details to the reader. □

Euclid's "extended" algorithm, i.e. the algorithm used to compute (u, v, d) and not d alone, is useful in many different contexts. For example, one frequent use is to compute an inverse (or more generally a division) modulo m. Assume one wants to compute the inverse of a number b modulo m. Then, using Algorithm 1.3.6, 1.3.7 or 1.3.8, compute (u, v, d) such that $bu + mv = d = (b, m)$. If $d > 1$ send an error message stating that b is not invertible, otherwise the inverse of b is u. Notice that in this case, we can avoid computing v in step 2 of Algorithm 1.3.6 and in the analogous steps in the other algorithms.

There are other methods to compute $b^{-1} \bmod m$ when the factorization of m is known, for example when m is a prime. By Euler-Fermat's Theorem

1.4.2, we know that, if $(b, m) = 1$ (which can be tested very quickly since the factorization of m is known), then

$$b^{\phi(m)} \equiv 1 \pmod{m},$$

where $\phi(m)$ is Euler's ϕ function (see [H-W]). Hence, the inverse of b modulo m can be obtained by computing

$$b^{-1} = b^{\phi(m)-1} \pmod{m},$$

using the powering Algorithm 1.2.1.

Note however that the powering algorithms are $O(\ln^3 m)$ algorithms, which is worse than the time for Euclid's extended algorithm. Nonetheless they can be useful in certain cases. A practical comparison of these methods is done in [Bre1].

1.3.3 The Chinese Remainder Theorem

We recall the following theorem:

Theorem 1.3.9 (Chinese Remainder Theorem). *Let $m_1, \ldots, m_k$ and $x_1, \ldots, x_k$ be integers. Assume that for every pair (i, j) we have*

$$x_i \equiv x_j \pmod{\gcd(m_i, m_j)}.$$

There exists an integer x such that

$$x \equiv x_i \pmod{m_i} \quad \text{for} \quad 1 \leq i \leq k.$$

Furthermore, x is unique modulo the least common multiple of $m_1, \ldots, m_k$.

Corollary 1.3.10. *Let $m_1, \ldots, m_k$ be pairwise coprime integers, i.e. such that*

$$\gcd(m_i, m_j) = 1 \qquad \text{when } i \neq j.$$

Then, for any integers x_i, there exists an integer x, unique modulo $\prod m_i$, such that

$$x \equiv x_i \pmod{m_i} \quad \text{for} \quad 1 \leq i \leq k.$$

We need an algorithm to compute x. We will consider only the case where the m_i are pairwise coprime, since this is by far the most useful situation. Set $M = \prod_{1 \leq i \leq k} m_i$ and $M_i = M/m_i$. Since the m_i are coprime in pairs, $\gcd(M_i, m_i) = 1$ hence by Euclid's extended algorithm we can find a_i such that $a_i M_i \equiv 1 \pmod{m_i}$. If we set

$$x = \sum_{1 \le i \le k} a_i M_i x_i,$$

it is clear that x satisfies the required conditions. Therefore, we can output $x \bmod M$ as the result.

This method could be written explicitly as a formal algorithm. However we want to make one improvement before doing so. Notice that the necessary constants a_i are small (less than m_i), but the M_i or the $a_i M_i$ which are also needed can be very large. There is an ingenious way to avoid using such large numbers, and this leads to the following algorithm. Its verification is left to the reader.

Algorithm 1.3.11 (Chinese). Given pairwise coprime integers m_i $(1 \le i \le k)$ and integers x_i, this algorithm finds an integer x such that $x \equiv x_i \pmod{m_i}$ for all i. Note that steps 1 and 2 are a precomputation which needs to be done only once when the m_i are fixed and the x_i vary.

1. [Initialize] Set $j \leftarrow 2$, $C_1 \leftarrow 1$. In addition, if it is not too costly, reorder the m_i (and hence the x_i) so that they are in increasing order.
2. [Precomputations] Set $p \leftarrow m_1 m_2 \cdots m_{j-1} \pmod{m_j}$. Compute (u, v, d) such that $up + vm_j = d = \gcd(p, m_j)$ using a suitable version of Euclid's extended algorithm. If $d > 1$ output an error message (the m_i are not pairwise coprime). Otherwise, set $C_j \leftarrow u$, $j \leftarrow j + 1$, and go to step 2 if $j \le k$.
3. [Compute auxiliary constants] Set $y_1 \leftarrow x_1 \bmod m_1$, and for $j = 2, \ldots, k$ compute (as written)

$$y_j \leftarrow (x_j - (y_1 + m_1(y_2 + m_2(y_3 + \cdots + m_{j-2}y_{j-1}) \cdots))C_j \bmod m_j.$$

4. [Terminate] Output

$$x \leftarrow y_1 + m_1(y_2 + m_2(y_3 + \cdots + m_{k-1}y_k) \cdots)),$$

and terminate the algorithm.

Note that we will have $0 \le x < M = \prod m_i$.

As an exercise, the reader can give an algorithm which finds x in the more general case of Theorem 1.3.9 where the m_i are not assumed to be pairwise coprime. It is enough to write an algorithm such as the one described before Algorithm 1.3.11, since it will not be used very often (Exercise 9).

Since this algorithm is more complex than the algorithm mentioned previously, it should only be used when the m_i are fixed moduli, and not just for a one shot problem. In this last case is it preferable to use the formula for two numbers inductively as follows. We want $x \equiv x_i \pmod{m_i}$ for $i = 1, 2$. Since the m_i are relatively prime, using Euclid's extended algorithm we can find u and v such that

$$um_1 + vm_2 = 1.$$

It is clear that

$$x = um_1x_2 + vm_2x_1 \bmod m_1m_2$$

is a solution to our problem. This leads to the following.

Algorithm 1.3.12 (Inductive Chinese). Given pairwise coprime integers m_i $(1 \leq i \leq k)$ and integers x_i, this algorithm finds an integer x such that $x \equiv x_i \pmod{m_i}$ for all i.

1. [Initialize] Set $i \leftarrow 1$, $m \leftarrow m_1$, $x \leftarrow x_1$.
2. [Finished?] If $i = k$ output x and terminate the algorithm. Otherwise, set $i \leftarrow i+1$, and by a suitable version of Euclid's extended algorithm compute u and v such that $um + vm_i = 1$.
3. [Compute next x] Set $x \leftarrow umx_i + vm_ix$, $m \leftarrow mm_i$, $x \leftarrow x \bmod m$ and go to step 2.

Note that the results and algorithms of this section remain true if we replace $\mathbb{Z}$ by any Euclidean domain, for example the polynomial ring $K[X]$ where K is a field.

1.3.4 Continued Fraction Expansions of Real Numbers

We now come to a subject which though closely linked to Euclid's algorithm, has a different flavor. Consider first the following apparently simple problem. Let $x \in \mathbb{R}$ be given by an approximation (for example a decimal or binary one). Decide if x is a rational number or not. Of course, this question as posed does not really make sense, since an approximation is usually itself a rational number. In practice however the question does make a lot of sense in many different contexts, and we can make it algorithmically more precise. For example, assume that one has an algorithm which allows us to compute x to as many decimal places as one likes (this is usually the case). Then, if one claims that x is (approximately) equal to a rational number p/q, this means that p/q should still be extremely close to x whatever the number of decimals asked for, p and q being fixed. This is still not completely rigorous, but it comes quite close to actual practice, so we shall be content with this notion.

Now how does one find p and q if x is indeed a rational number? The standard (and algorithmically excellent) answer is to compute the *continued fraction expansion* of x, i.e. find integers a_i such that $a_i \geq 1$ for $i \geq 1$ and

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \ddots}}},$$

which we shall write as $x = [a_0, a_1, a_2, a_3, \ldots]$. If a/b is the given (rational) approximation to x, then the a_i are obtained by simply using Euclid's algorithm

on the pair (a, b), the a_i being the successive partial quotients. The number x is rational if and only if its continued fraction expansion is finite, i.e. if and only if one of the a_i is infinite. Since x is only given with the finite precision a/b, x will be considered rational if x has a very large partial quotient a_i in its continued fraction expansion. Of course this is subjective, but should be put to the stringent test mentioned above. For example, if one uses the approximation $\pi \approx 3.1415926$ one finds that the continued fraction for π should start with $[3, 7, 15, 1, 243, \ldots]$ and 243 does seem a suspiciously large partial quotient, so we suspect that $\pi = 355/113$, which is the rational number whose continued fraction is exactly $[3, 7, 15, 1]$. If we compute a few more decimals of π however, we see that this equality is not true. Nonetheless, $355/113$ is still an excellent approximation to π (the continued fraction expansion of π starts in fact $[3, 7, 15, 1, 292, 1, \ldots]$).

To implement a method for computing continued fractions of real numbers, I suggest using the following algorithm, which says exactly when to stop.

Algorithm 1.3.13 (Lehmer). Given a real number x by two rational numbers a/b and a'/b' such that $a/b \leq x \leq a'/b'$, this algorithm computes the continued fraction expansion of x and stops exactly when it is not possible to determine the next partial quotient from the given approximants a/b and a'/b', and it gives lower and upper bounds for this next partial quotient.

1. [Initialize] Set $i \leftarrow 0$.
2. [Euclidean step] Let $a = bq + r$ the Euclidean division of a by b, and set $r' \leftarrow a' - b'q$. If $r' < 0$ or $r' \geq b'$ set $q' \leftarrow \lfloor a'/b' \rfloor$ and go to step 4.
3. [Output quotient] Set $a_i \leftarrow q$ and output a_i, then set $i \leftarrow i+1$, $a \leftarrow b$, $b \leftarrow r$, $a' \leftarrow b'$ and $b' \leftarrow r'$. If b and b' are non-zero, go to step 2. If $b = b' = 0$, terminate the algorithm. Finally, if $b = 0$ set $q \leftarrow \infty$ and $q' \leftarrow \lfloor a'/b' \rfloor$ while if $b' = 0$ set $q \leftarrow \lfloor a/b \rfloor$ and $q' \leftarrow \infty$.
4. [Terminate] If $q > q'$ output the inequality $q' \leq a_i \leq q$, otherwise output $q \leq a_i \leq q'$. Terminate the algorithm.

Note that the ∞ mentioned in step 3 is only a mathematical abstraction needed to make step 4 make sense, but it does not need to be represented in a machine by anything more than some special code.

This algorithm runs in at most twice the time needed for the Euclidean algorithm on a and b alone, since, in addition to doing one Euclidean division at each step, we also multiply q by b'.

We can now solve the following problem: given two complex numbers z_1 and z_2, are they $\mathbb{Q}$-linearly dependent? This is equivalent to z_1/z_2 being rational, so the solution is this: compute $z \leftarrow z_1/z_2$. If the imaginary part of z is non-zero (to the degree of approximation that one has), then z_1 and z_2 are not even $\mathbb{R}$-linearly dependent. If it is zero, then compute the continued fraction expansion of the real part of z using algorithm 1.3.13, and look for large partial quotients as explained above.

We will see in Section 2.7.2 that the LLL algorithms allow us to determine in a satisfactory way the problem of $\mathbb{Q}$-linear dependence of more than two complex or real numbers.

Another closely related problem is the following: given two vectors $\mathbf{a}$ and $\mathbf{b}$ in a Euclidean vector space, determine the shortest non-zero vector which is a $\mathbb{Z}$-linear combination of $\mathbf{a}$ and $\mathbf{b}$ (we will see in Chapter 2 that the set of such $\mathbb{Z}$-linear combinations is called a *lattice*, here of dimension 2). One solution, called Gaussian reduction, is again a form of Euclid's algorithm, and is as follows.

Algorithm 1.3.14 (Gauss). Given two linearly independent vectors $\mathbf{a}$ and $\mathbf{b}$ in a Euclidean vector space, this algorithm determines one of the shortest non-zero vectors which is a $\mathbb{Z}$-linear combination of $\mathbf{a}$ and $\mathbf{b}$. We denote by $\cdot$ the Euclidean inner product and write $|\mathbf{a}|^2 = \mathbf{a} \cdot \mathbf{a}$. We use a temporary scalar variable T, and a temporary vector variable $\mathbf{t}$.

1. [Initialize] Set $A \leftarrow |\mathbf{a}|^2$, $B \leftarrow |\mathbf{b}|^2$. If $A < B$ then exchange $\mathbf{a}$ and $\mathbf{b}$ and exchange A and B.
2. [Euclidean step] Set $n \leftarrow \mathbf{a} \cdot \mathbf{b}$, $r \leftarrow \lfloor n/B \rceil$, where $\lfloor x \rceil = \lfloor x + 1/2 \rfloor$ is the nearest integer to x, and $T \leftarrow A - 2rn + r^2 B$.
3. [Finished?] If $T \geq B$ then output $\mathbf{b}$ and terminate the algorithm. Otherwise, set $\mathbf{t} \leftarrow \mathbf{a} - r\mathbf{b}$, $\mathbf{a} \leftarrow \mathbf{b}$, $\mathbf{b} \leftarrow \mathbf{t}$, $A \leftarrow B$, $B \leftarrow T$ and go to step 2.

Proof. Note that A and B are always equal to $|\mathbf{a}|^2$ and $|\mathbf{b}|^2$ respectively. I first claim that an integer r such that $|\mathbf{a} - r\mathbf{b}|$ has minimal length is given by the formula of step 2. Indeed, we have

$$|\mathbf{a} - x\mathbf{b}|^2 = Bx^2 - 2\mathbf{a} \cdot \mathbf{b}x + A,$$

and this is minimum for *real* x for $x = \mathbf{a} \cdot \mathbf{b}/B$. Hence, since a parabola is symmetrical at its minimum, the minimum for integral x is the nearest integer (or one of the two nearest integers) to the minimum, and this is the formula given in step 2.

Thus, at the end of the algorithm we know that $|\mathbf{a} - m\mathbf{b}| \geq |\mathbf{b}|$ for all integers m. It is clear that the transformation which sends the pair $(\mathbf{a}, \mathbf{b})$ to the pair $(\mathbf{b}, \mathbf{a} - r\mathbf{b})$ has determinant -1, hence the $\mathbb{Z}$-module L generated by $\mathbf{a}$ and $\mathbf{b}$ stays the same during the algorithm. Therefore, let $\mathbf{x} = u\mathbf{a} + v\mathbf{b}$ be a non-zero element of L. If $u = 0$, we must have $v \neq 0$ hence trivially $|\mathbf{x}| \geq |\mathbf{b}|$. Otherwise, let $v = uq + r$ be the Euclidean division of v by u, where $0 \leq r < |u|$. Then we have

$$|\mathbf{x}| = |u(\mathbf{a} + q\mathbf{b}) + r\mathbf{b}| \geq |u||\mathbf{a} + q\mathbf{b}| - |r||\mathbf{b}| \geq (|u| - |r|)|\mathbf{b}| \geq |\mathbf{b}|$$

since by our above claim $|\mathbf{a} + q\mathbf{b}| \geq |\mathbf{b}|$ for any integer q, hence $\mathbf{b}$ is indeed one of the shortest vectors of L, proving the validity of the algorithm.

Note that the algorithm must terminate since there are only a finite number of vectors of L with norm less than or equal to a given constant (compact+discrete=finite!). In fact the number of steps can easily be seen to be comparable to that of the Euclidean algorithm, hence this algorithm is very efficient. □

We will see in Section 2.6 that the LLL algorithm allows us to determine efficiently small $\mathbb{Z}$-linear combinations for more than two linearly independent vectors in a Euclidean space. It does not always give an optimal solution, but, in most situations, the results are sufficiently good to be very useful.

1.4 The Legendre Symbol

1.4.1 The Groups $(\mathbb{Z}/n\mathbb{Z})^*$

By definition, when A is a commutative ring with unit, we will denote by A^* the group of *units* of A, i.e. of invertible elements of A. It is clear that A^* is a group, and also that $A^* = A \setminus \{0\}$ if and only if A is a field. Now we have the following fundamental theorem which gives the structure of $(\mathbb{Z}/n\mathbb{Z})^*$ (see [Ser] and Exercise 13).

Theorem 1.4.1. *We have*

$$|(\mathbb{Z}/n\mathbb{Z})^*| = \phi(n) = n\prod_{p|n}\left(1-\frac{1}{p}\right),$$

and more precisely

$$(\mathbb{Z}/n\mathbb{Z})^* \simeq \prod_{p^\alpha \| n} (\mathbb{Z}/p^\alpha\mathbb{Z})^*,$$

where

$$(\mathbb{Z}/p^\alpha\mathbb{Z})^* \simeq \mathbb{Z}/(p-1)p^{\alpha-1}\mathbb{Z}$$

(i.e. is cyclic) when $p \geq 3$ or $p = 2$ and $\alpha \leq 2$, and

$$(\mathbb{Z}/2^\alpha\mathbb{Z})^* \simeq \mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/2^{\alpha-2}\mathbb{Z}$$

when $p = 2$ and $\alpha \geq 3$.

Now when $(\mathbb{Z}/n\mathbb{Z})^*$ is cyclic, i.e. by the above theorem when n is equal either to p^α, $2p^\alpha$ with p an odd prime, or $n = 2$ or 4, an integer g such that the class of g generates $(\mathbb{Z}/n\mathbb{Z})^*$ will be called a *primitive root* modulo n. Recall that the *order* of an element g in a group is the least positive integer n such that g^n is equal to the identity element of the group. When the group is finite, the order of any element divides the order of the group. Furthermore, g is a

primitive root of $(\mathbb{Z}/n\mathbb{Z})^*$ if and only if its order is exactly equal to $\phi(n)$. As a corollary of the above results, we obtain the following:

Proposition 1.4.2.

(1) (Fermat). *If p is a prime and a is not divisible by p, then we have*

$$a^{p-1} \equiv 1 \pmod{p}.$$

(2) (Euler). *More generally, if n is a positive integer, then for any integer a coprime to n we have*

$$a^{\phi(n)} \equiv 1 \pmod{n},$$

and even

$$a^{\phi(n)/2} \equiv 1 \pmod{n}$$

if n is not equal to 2, 4, p^α or $2p^\alpha$ with p an odd prime.

To compute the order of an element in a finite group G, we use the following straightforward algorithm.

Algorithm 1.4.3 (Order of an Element). Given a finite group G of cardinality $h = |G|$, and an element $g \in G$, this algorithm computes the order of g in G. We denote by 1 the unit element of G.

1. [Initialize] Compute the prime factorization of h, say $h = p_1^{v_1} p_2^{v_2} \cdots p_k^{v_k}$, and set $e \leftarrow h$, $i \leftarrow 0$.
2. [Next p_i] Set $i \leftarrow i + 1$. If $i > k$, output e and terminate the algorithm. Otherwise, set $e \leftarrow e/p_i^{v_i}$, $g_1 \leftarrow g^e$.
3. [Compute local order] While $g_1 \neq 1$, set $g_1 \leftarrow g_1^{p_i}$ and $e \leftarrow e \cdot p_i$. Go to step 2.

Note that we need the complete factorization of h for this algorithm to work. This may be difficult when the group is very large.

Let p be a prime. To find a primitive root modulo p there seems to be no better way than to proceed as follows. Try $g = 2$, $g = 3$, etc ... until g is a primitive root. One should avoid perfect powers since if $g = g_0^k$, then if g is a primitive root, so is g_0 which has already been tested.

To see whether g is a primitive root, we could compute the order of g using the above algorithm. But it is more efficient to proceed as follows.

Algorithm 1.4.4 (Primitive Root). Given an odd prime p, this algorithm finds a primitive root modulo p.

1. [Initialize a] Set $a \leftarrow 1$ and let $p - 1 = p_1^{v_1} p_2^{v_2} \cdots p_k^{v_k}$ be the complete factorization of $p - 1$.

2. [Initialize check] Set $a \leftarrow a+1$ and $i \leftarrow 1$.
3. [Check p_i] Compute $e \leftarrow a^{(p-1)/p_i}$. If $e = 1$ go to step 2. Otherwise, set $i \leftarrow i+1$.
4. [finished?] If $i > k$ output a and terminate the algorithm, otherwise go to step 3.

Note that we do not avoid testing prime powers, hence this simple algorithm can still be improved if desired. In addition, the test for $p_i = 2$ can be replaced by the more efficient check that the Legendre symbol $\left(\frac{a}{p}\right)$ is equal to -1 (see Algorithm 1.4.10 below).

If n is not a prime, but is such that there exists a primitive root modulo n, we could, of course, use the above two algorithms by modifying them suitably. It is more efficient to proceed as follows.

First, if $n = 2$ or $n = 4$, $g = n-1$ is a primitive root. When $n = 2^a$ is a power of 2 with $a \geq 3$, $(\mathbb{Z}/n\mathbb{Z})^*$ is not cyclic any more, but is isomorphic to the product of $\mathbb{Z}/2\mathbb{Z}$ with a cyclic group of order 2^{a-2}. Then $g = 5$ is always a generator of this cyclic subgroup (see Exercise 14), and can serve as a substitute in this case if needed.

When $n = p^a$ is a power of an odd prime, with $a \geq 2$, then we use the following lemma.

Lemma 1.4.5. *Let p be an odd prime, and let g be a primitive root modulo p. Then either g or $g+p$ is a primitive root modulo every power of p.*

Proof. For any m we have $m^p \equiv m \pmod{p}$, hence it follows that for every prime l dividing $p-1$, $g^{p^{a-1}(p-1)/l} \equiv g^{(p-1)/l} \not\equiv 1 \pmod{p}$. So for g to be a primitive root, we need only that $g^{p^{a-2}(p-1)} \not\equiv 1 \pmod{p^a}$. But one checks immediately by induction that $x^p \equiv 1 \pmod{p^a}$ implies that $x \equiv 1 \pmod{p^b}$ for every $b \leq a-1$. Applying this to $x = g^{p^{a-2}(p-1)}$ we see that our condition on g is equivalent to the same condition with a replaced by $a-1$, hence by induction to the condition $g^{p-1} \not\equiv 1 \pmod{p^2}$. But if $g^{p-1} \equiv 1 \pmod{p^2}$, then by the binomial theorem $(g+p)^{p-1} \equiv 1 - pg^{p-2} \not\equiv 1 \pmod{p^2}$, thus proving the lemma. □

Therefore to find a primitive root modulo p^a for p an odd prime and $a \geq 2$, proceed as follows: first compute g a primitive root modulo p using Algorithm 1.4.4, then compute $g_1 = g^{p-1} \bmod p^2$. If $g_1 \neq 1$, g is a primitive root modulo p^a for every a, otherwise $g+p$ is.

Finally, note that when p is an odd prime, if g is a primitive root modulo p^a then g or $g+p^a$ (whichever is odd) is a primitive root modulo $2p^a$.

1.4.2 The Legendre-Jacobi-Kronecker Symbol

Let p be an odd prime. Then it is easy to see that for a given integer a, the congruence

$$x^2 \equiv a \pmod{p}$$

can have either no solution (we say in this case that a is a quadratic non-residue mod p), one solution if $a \equiv 0 \pmod{p}$, or two solutions (we then say that a is a quadratic residue mod p). Define the Legendre symbol $\left(\frac{a}{p}\right)$ as being -1 if a is a quadratic non-residue, 0 if $a = 0$, and 1 if a is a quadratic residue. Then the number of solutions modulo p of the above congruence is $1 + \left(\frac{a}{p}\right)$. Furthermore, one can easily show that this symbol has the following properties (see e.g. [H-W]):

Proposition 1.4.6.

(1) *The Legendre symbol is multiplicative, i.e.*

$$\left(\frac{a}{p}\right)\left(\frac{b}{p}\right) = \left(\frac{ab}{p}\right).$$

In particular, the product of two quadratic non-residues is a quadratic residue.

(2) *We have the congruence*

$$a^{(p-1)/2} \equiv \left(\frac{a}{p}\right) \pmod{p}.$$

(3) *There are as many quadratic residues as non-residues mod p, i.e. $(p-1)/2$.*

We will see that the Legendre symbol is fundamental in many problems. Thus, we need a way to compute it. One idea is to use the congruence $a^{(p-1)/2} \equiv \left(\frac{a}{p}\right) \pmod{p}$. Using the powering Algorithm 1.2.1, this enables us to compute the Legendre symbol in time $O(\ln^3 p)$. We can improve on this by using the Legendre-Gauss quadratic reciprocity law, which is itself a result of fundamental importance:

Theorem 1.4.7. *Let p be an odd prime. Then:*

(1)

$$\left(\frac{-1}{p}\right) = (-1)^{(p-1)/2}, \quad \left(\frac{2}{p}\right) = (-1)^{(p^2-1)/8}.$$

(2) *If q is an odd prime different from p, then we have the reciprocity law:*

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{(p-1)(q-1)/4}.$$

For a proof, see Exercises 16 and 18 and standard textbooks (e.g. [H-W], [Ire-Ros]).

This theorem can certainly help us to compute Legendre symbols since $\left(\frac{a}{p}\right)$ is multiplicative in a and depends only on a modulo p. A direct use of Theorem 1.4.7 would require factoring all the numbers into primes, and this is very slow. Luckily, there is an extension of this theorem which takes care of this problem. We first need to extend the definition of the Legendre symbol.

Definition 1.4.8. *We define the Kronecker (or Kronecker-Jacobi) symbol* $\left(\frac{a}{b}\right)$ *for any* a *and* b *in* $\mathbb{Z}$ *in the following way.*

(1) *If* $b = 0$, *then* $\left(\frac{a}{0}\right) = 1$ *if* $a = \pm 1$, *and is equal to* 0 *otherwise.*
(2) *For* $b \neq 0$, *write* $b = \prod p$, *where the* p *are not necessarily distinct primes (including* $p = 2$*), or* $p = -1$ *to take care of the sign. Then we set*

$$\left(\frac{a}{b}\right) = \prod \left(\frac{a}{p}\right),$$

where $\left(\frac{a}{p}\right)$ *is the Legendre symbol defined above for* $p > 2$, *and where we define*

$$\left(\frac{a}{2}\right) = \begin{cases} 0, & \text{if } a \text{ is even} \\ (-1)^{(a^2-1)/8}, & \text{if } a \text{ is odd.} \end{cases}$$

and also

$$\left(\frac{a}{-1}\right) = \begin{cases} 1, & \text{if } a \geq 0 \\ -1, & \text{if } a < 0. \end{cases}$$

Then, from the properties of the Legendre symbol, and in particular from the reciprocity law 1.4.7, one can prove that the Kronecker symbol has the following properties:

Theorem 1.4.9.

(1) $\left(\frac{a}{b}\right) = 0$ *if and only if* $(a, b) \neq 1$
(2) *For all* a, b *and* c *we have*

$$\left(\frac{ab}{c}\right) = \left(\frac{a}{c}\right)\left(\frac{b}{c}\right), \quad \left(\frac{a}{bc}\right) = \left(\frac{a}{b}\right)\left(\frac{a}{c}\right) \quad \text{if } bc \neq 0$$

(3) $b > 0$ *being fixed, the symbol* $\left(\frac{a}{b}\right)$ *is periodic in* a *of period* b *if* $b \not\equiv 2 \pmod 4$, *otherwise it is periodic of period* $4b$.
(4) $a \neq 0$ *being fixed (positive or negative), the symbol* $\left(\frac{a}{b}\right)$ *is periodic in* b *of period* $|a|$ *if* $a \equiv 0$ *or* $1 \pmod 4$, *otherwise it is periodic of period* $4|a|$.
(5) *The formulas of Theorem 1.4.7 are still true if* p *and* q *are only supposed to be positive odd integers, not necessarily prime.*

Note that in this theorem (as in the rest of this book), when we say that a function $f(x)$ is periodic of period b, this means that for all x, $f(x+b) = f(x)$, but b need not be the smallest possible period.

Theorem 1.4.9 is a necessary prerequisite for any study of quadratic fields, and the reader is urged to prove it by himself (Exercise 17).

As has been mentioned, a consequence of this theorem is that it is easy to design a fast algorithm to compute Legendre symbols, and more generally Kronecker symbols if desired.

Algorithm 1.4.10 (Kronecker). Given $a, b \in \mathbb{Z}$, this algorithm computes the Kronecker symbol $\left(\frac{a}{b}\right)$ (hence the Legendre symbol when b is an odd prime).

1. [Test b equal to 0] If $b = 0$ then output 0 if $|a| \neq 1$, 1 if $|a| = 1$ and terminate the algorithm.

2. [Remove 2's from b] If a and b are both even, output 0 and terminate the algorithm. Otherwise, set $v \leftarrow 0$ and while b is even set $v \leftarrow v + 1$ and $b \leftarrow b/2$. Then if v is even set $k \leftarrow 1$, otherwise set $k \leftarrow (-1)^{(a^2-1)/8}$ (by table lookup, *not* by computing $(a^2-1)/8$). Finally if $b < 0$ set $b \leftarrow -b$, and if in addition $a < 0$ set $k \leftarrow -k$.

3. [Finished?] (Here b is odd and $b > 0$.) If $a = 0$ then output 0 if $b > 1$, k if $b = 1$, and terminate the algorithm. Otherwise, set $v \leftarrow 0$ and while a is even do $v \leftarrow v + 1$ and $a \leftarrow a/2$. If v is odd set $k \leftarrow (-1)^{(b^2-1)/8}k$.

4. [Apply reciprocity] Set

$$k \leftarrow (-1)^{(a-1)(b-1)/4}k,$$

(using if statements and no multiplications), and then $r \leftarrow |a|$, $a \leftarrow b \bmod r$, $b \leftarrow r$ and go to step 3.

Remarks.

(1) As mentioned, the expressions $(-1)^{(a^2-1)/8}$ and $(-1)^{(a-1)(b-1)/4}$ should not be computed as powers, even though they are written this way. For example, to compute the first expression, set up and save a table `tab2` containing

$$\{0, 1, 0, -1, 0, -1, 0, 1\},$$

and then the formula $(-1)^{(a^2-1)/8} =$ `tab2[a&7]`, the `&` symbol denoting bitwise and, which is a very fast operation compared to multiplication (note that `a&7` is equivalent to a mod 8). The instruction $k \leftarrow (-1)^{(a-1)(b-1)/4}k$ is very efficiently translated in C by

```
if(a&b&2) k= -k;
```

(2) We need to prove that the algorithm is valid! It terminates since, because except possibly the first time, at the beginning of step 3 we have $0 < b < a$ and the value of b is strictly decreasing. It gives the correct result because of the following lemma which is an immediate corollary of Theorem 1.4.9:

Lemma 1.4.11. *If a and b are odd integers with $b > 0$ (but not necessarily $a > 0$), then we have*

$$\left(\frac{a}{b}\right) = (-1)^{(a-1)(b-1)/4}\left(\frac{b}{|a|}\right).$$

(3) We may want to avoid cleaning out the powers of 2 in step 3 at each pass through the loop. We can do this by slightly changing step 4 so as to always end up with an odd value of a. This however may have disastrous effects on the running time, which may become exponential instead of polynomial time (see [Bac-Sha] and Exercise 24).

Note that Algorithm 1.4.10 can be slightly improved (by a small constant factor) by adding the following statement at the end of the assignments of step 4, before going back to step 3: If $a > r/2$, then $a = a - r$. This simply means that we ask, not for the residue of $a \bmod r$ which is between 0 and $r - 1$, but for the one which is least in absolute value, i.e. between $-r/2$ and $r/2$. This modification could also be used in Euclid's algorithms if desired, if tests suggest that it is faster in practice.

One can also use the binary version of Euclid's algorithm to compute Kronecker symbols. Since, in any case, the prime 2 plays a special role, this does not really increase the complexity, and gives the following algorithm.

Algorithm 1.4.12 (Kronecker-Binary). Given $a, b \in \mathbb{Z}$, this algorithm computes the Kronecker symbol $\left(\frac{a}{b}\right)$ (hence the Legendre symbol when b is an odd prime).

1. [Test $b = 0$] If $b = 0$ then output 0 if $|a| \neq 1$, 1 if $|a| = 1$ and terminate the algorithm.
2. [Remove 2's from b] If a and b are both even, output 0 and terminate the algorithm. Otherwise, set $v \leftarrow 0$ and while b is even set $v \leftarrow v + 1$ and $b \leftarrow b/2$. Then if v is even set $k \leftarrow 1$, otherwise set $k \leftarrow (-1)^{(a^2-1)/8}$ (by table lookup, *not* by computing $(a^2 - 1)/8$). Finally, if $b < 0$ set $b \leftarrow -b$, and if in addition $a < 0$ set $k \leftarrow -k$.
3. [Reduce size once] (Here b is odd and $b > 0$.) Set $a \leftarrow a \bmod b$.
4. [Finished?] If $a = 0$, output 0 if $b > 1$, k if $b = 1$, and terminate the algorithm.
5. [Remove powers of 2] Set $v \leftarrow 0$ and, while a is even, set $v \leftarrow v + 1$ and $a \leftarrow a/2$. If v is odd, set $k \leftarrow (-1)^{(b^2-1)/8}k$.
6. [Subtract and apply reciprocity] (Here a and b are odd.) Set $r \leftarrow b - a$. If $r > 0$, then set $k \leftarrow (-1)^{(a-1)(b-1)/4}k$ (using if statements), $b \leftarrow a$ and $a \leftarrow r$, else set $a \leftarrow -r$. Go to step 4.

Note that we cannot immediately reduce a modulo b at the beginning of the algorithm. This is because when b is even the Kronecker symbol $\left(\frac{a}{b}\right)$ is not

periodic of period b in general, but only of period $4b$. Apart from this remark, the proof of the validity of this algorithm follows immediately from Theorem 1.4.10 and the validity of the binary algorithm. □

The running time of all of these Legendre symbol algorithms has the same order of magnitude as Euclid's algorithm, i.e. $O(\ln^2 N)$ when carefully programmed, where N is an upper bound on the size of the inputs a and b. Note however that the constants will be different because of the special treatment of even numbers.

1.5 Computing Square Roots Modulo p

We now come to a slightly more specialized question. Let p be an odd prime number, and suppose that we have just checked that $\left(\frac{a}{p}\right) = 1$ using one of the algorithms given above. Then by definition, there exists an x such that $x^2 \equiv a \pmod{p}$. How do we find x? Of course, a brute force search would take time $O(p)$ and, even for p moderately large, is out of the question. We need a faster algorithm to do this. At this point the reader might want to try and find one himself before reading further. This would give a feel for the difficulty of the problem. (Note that we will be considering much more difficult and general problems later on, so it is better to start with a simple one.)

There is an easy solution which comes to mind that works for half of the primes p, i.e. primes $p \equiv 3 \pmod{4}$. I claim that in this case a solution is given by

$$x = a^{(p+1)/4} \pmod{p},$$

the computation being done using the powering Algorithm 1.2.1. Indeed, since a is a quadratic residue, we have $a^{(p-1)/2} \equiv 1 \pmod{p}$ hence

$$x^2 \equiv a^{(p+1)/2} \equiv a \cdot a^{(p-1)/2} \equiv a \pmod{p}$$

as claimed.

A less trivial solution works for half of the remaining primes, i.e. primes $p \equiv 5 \pmod{8}$. Since we have $a^{(p-1)/2} \equiv 1 \pmod{p}$ and since $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ is a field, we must have

$$a^{(p-1)/4} \equiv \pm 1 \pmod{p}.$$

Now, if the sign is $+$, then the reader can easily check as above that

$$x = a^{(p+3)/8} \pmod{p}$$

is a solution. Otherwise, using $p \equiv 5 \pmod{8}$ and Theorem 1.4.7, we know that $2^{(p-1)/2} \equiv -1 \pmod{p}$. Then one can check that

$$x = 2a \cdot (4a)^{(p-5)/8} \pmod{p}$$

is a solution.

Thus the only remaining case is $p \equiv 1 \pmod 8$. Unfortunately, this is the hardest case. Although, by methods similar to the one given above, one could give an infinite number of families of solutions, this would not be practical in any sense.

1.5.1 The Algorithm of Tonelli and Shanks

There are essentially three algorithms for solving the above problem. One is a special case of a general method for factoring polynomials modulo p, which we will study in Chapter 3. Another is due to Schoof and it is the only non-probabilistic polynomial time algorithm known for this problem. It is quite complex since it involves the use of elliptic curves (see Chapter 7), and its practicality is not clear, although quite a lot of progress has been achieved by Atkin. Therefore, we will not discuss it here. The third and last algorithm is due to Tonelli and Shanks, and although probabilistic, it is quite efficient. It is the most natural generalization of the special cases studied above. We describe this algorithm here.

We can always write

$$p - 1 = 2^e \cdot q, \quad \text{with } q \text{ odd.}$$

The multiplicative group $(\mathbb{Z}/p\mathbb{Z})^*$ is isomorphic to the (additive) group $\mathbb{Z}/(p-1)\mathbb{Z}$, hence its 2-Sylow subgroup G is a cyclic group of order 2^e. Assume that one can find a generator z of G. The squares in G are the elements of order dividing 2^{e-1}, and are also the even powers of z. Hence, if a is a quadratic residue mod p, then, since

$$a^{(p-1)/2} = (a^q)^{(2^{e-1})} \equiv 1 \pmod p,$$

$b = a^q \bmod p$ is a square in G, so there exists an even integer k with $0 \leq k < 2^e$ such that

$$a^q z^k = 1 \quad \text{in } G.$$

If one sets

$$x = a^{(q+1)/2} z^{k/2},$$

it is clear that $x^2 \equiv a \pmod p$, hence x is the answer. To obtain an algorithm, we need to solve two problems: finding a generator z of G, and computing the exponent k. Although very simple to solve in practice, the first problem is the probabilistic part of the algorithm. The best way to find z is as follows: choose at random an integer n, and compute $z = n^q \bmod p$. Then it is clear that z is a generator of G (i.e. $z^{2^{e-1}} = -1$ in G) if and only if n is a quadratic non-residue mod p, and this occurs with probability close to $1/2$ (exactly $(p-1)/(2p)$). Therefore, in practice, we will find a non-residue very quickly. For example, the probability that one does not find one after 20 trials is lower than 10^{-6}.

Finding the exponent k is slightly more difficult, and in fact is not needed explicitly (only $a^{(q+1)/2}z^{k/2}$ is needed). The method is explained in the following complete algorithm, which in this form is due to Shanks.

Algorithm 1.5.1 (Square Root Mod p). Let p be an odd prime, and $a \in \mathbb{Z}$. Write $p-1 = 2^e \cdot q$ with q odd. This algorithm, either outputs an x such that $x^2 \equiv a \pmod{p}$, or says that such an x does not exist (i.e. that a is a quadratic non-residue mod p).

1. [Find generator] Choose numbers n at random until $\left(\frac{n}{p}\right) = -1$. Then set $z \leftarrow n^q \pmod{p}$.
2. [Initialize] Set $y \leftarrow z$, $r \leftarrow e$, $x \leftarrow a^{(q-1)/2} \pmod{p}$, $b \leftarrow ax^2 \pmod{p}$, $x \leftarrow ax \pmod{p}$.
3. [Find exponent] If $b \equiv 1 \pmod{p}$, output x and terminate the algorithm. Otherwise, find the smallest $m \geq 1$ such that $b^{2^m} \equiv 1 \pmod{p}$. If $m = r$, output a message saying that a is not a quadratic residue mod p.
4. [Reduce exponent] Set $t \leftarrow y^{2^{r-m-1}}$, $y \leftarrow t^2$, $r \leftarrow m$, $x \leftarrow xt$, $b \leftarrow by$ (all operations done modulo p), and go to step 3.

Note that at the beginning of step 3 we always have the congruences modulo p:

$$ab \equiv x^2, \qquad y^{2^{r-1}} \equiv -1, \qquad b^{2^{r-1}} \equiv 1.$$

If G_r is the subgroup of G whose elements have an order dividing 2^r, then this says that y is a generator of G_r and that b is in G_{r-1}, in other words that b is a square in G_r. Since r is strictly decreasing at each loop of the algorithm, the number of loops is at most e. When $r \leq 1$ we have $b = 1$ hence the algorithm terminates, and the above congruence shows that x is one of the square roots of a mod p.

It is easy to show that, on average, steps 3 and 4 will require $e^2/4$ multiplications mod p, and at most e^2. Hence the expected running time of this algorithm is $O(\ln^4 p)$. □

Remarks.

(1) In the algorithm above, we have not explicitly computed the value of the exponent k such that $a^q z^k = 1$ but it is easy to do so if needed (see Exercise 25).

(2) As already mentioned, Shanks's algorithm is probabilistic, although the only non-deterministic part is finding a quadratic non-residue mod p, which seems quite a harmless task. One could try making it completely deterministic by successively trying $n = 2, 3 \ldots$ in step 1 until a non-residue is found. This is a reasonable method, but unfortunately the most powerful analytical tools only allow us to prove that the smallest quadratic non-residue is $O(p^\alpha)$ for a non-zero α. Thus, this deterministic algorithm,

although correct, may have, as far as we know, an exponential running time.

If one assumes the Generalized Riemann Hypothesis (GRH), then one can prove much more, i.e. that the smallest quadratic non-residue is $O(\ln^2 p)$, hence this gives a polynomial running time (in $O(\ln^4 p)$ since computing a Legendre symbol is in $O(\ln^2 p)$). In fact, Bach [Bach] has proved that for $p > 1000$ the smallest non-residue is less than $2\ln^2 p$. In any case, in practice the probabilistic method and the sequential method (i.e. choosing $n = 2, 3, \ldots$) give essentially equivalent running times.

(3) If m is an integer whose factorization into a product of prime powers is completely known, it is easy to write an algorithm to solve the more general problem $x^2 \equiv a \pmod{m}$ (see Exercise 30).

1.5.2 The Algorithm of Cornacchia

A well known theorem of Fermat (see [H-W]) says that an odd prime p is a sum of two squares if and only if $p \equiv 1 \bmod 4$, i.e. if and only if -1 is a quadratic residue mod p. Furthermore, up to sign and exchange, the representation of p as a sum of two squares is unique. Thus, it is natural to ask for an algorithm to compute x and y such that $x^2 + y^2 = p$ when $p \equiv 1 \bmod 4$. More generally, given a positive integer d and an odd prime p, one can ask whether the equation

$$x^2 + dy^2 = p$$

has a solution, and for an algorithm to find x and y when they exist. There is a pretty algorithm due to Cornacchia which solves both problems simultaneously. For the beautiful and deep *theory* concerning the first problem, which is closely related to complex multiplication (see Section 7.2) see [Cox].

First, note that a necessary condition for the existence of a solution is that $-d$ be a quadratic residue modulo p. Indeed, we clearly must have $y \not\equiv 0 \bmod p$ hence

$$(xy^{-1})^2 \equiv -d \bmod p,$$

where y^{-1} denotes the inverse of y modulo p. We therefore assume that this condition is satisfied. By using Algorithm 1.5.1 we can find an integer x_0 such that

$$x_0^2 \equiv -d \bmod p$$

and we may assume that $p/2 < x_0 < p$. Cornacchia's algorithm tells us that we should simply apply Euclid's Algorithm 1.3.1 to the pair $(a, b) = (p, x_0)$ until we obtain a number b such that $b < \sqrt{p}$. Then we set $c \leftarrow (p - b^2)/d$, and if c is the square of an integer s, the equation $x^2 + dy^2 = p$ has $(x, y) = (b, s)$ as (essentially unique) solution, otherwise it has no solution. This leads to the following algorithm.

Algorithm 1.5.2 (Cornacchia). Let p be a prime number and d be an integer such that $0 < d < p$. This algorithm either outputs an integer solution (x, y) to

the Diophantine equation $x^2+dy^2=p$, or says that such a solution does not exist.

1. [Test if residue] Using Algorithm 1.4.12 compute $k \leftarrow \left(\frac{-d}{p}\right)$. If $k=-1$, say that the equation has no solution and terminate the algorithm.

2. [Compute square root] Using Shanks's Algorithm 1.5.1, compute an integer x_0 such that $x_0^2 \equiv -d \bmod p$, and change x_0 into $\pm x_0 + kp$ so that $p/2 < x_0 < p$. Then set $a \leftarrow p$, $b \leftarrow x_0$ and $l \leftarrow \lfloor \sqrt{p} \rfloor$.

3. [Euclidean algorithm] If $b > l$, set $r \leftarrow a \bmod b$, $a \leftarrow b$, $b \leftarrow r$ and go to step 3.

4. [Test solution] If d does not divide $p-b^2$ or if $c=(p-b^2)/d$ is not the square of an integer (see Algorithm 1.7.3), say that the equation has no solution and terminate the algorithm. Otherwise, output $(x,y)=(b,\sqrt{c})$ and terminate the algorithm.

Let us give a numerical example. Assume that we want to solve $x^2+2y^2=97$. In step 1, we first compute $\left(\frac{-2}{97}\right)$ by Algorithm 1.4.12 (or directly since here it is easy), and find that -2 is a quadratic residue mod 97. Thus the equation may have a solution (and in fact it must have one since the class number of the ring of integers of $\mathbb{Q}(\sqrt{2})$ is equal to 1, see Chapter 5). In step 2, we compute x_0 such that $x_0^2 \equiv -2 \bmod 97$ using Algorithm 1.5.1. Using $n=5$ hence $z=28$, we readily find $x_0=17$. Then the Euclidean algorithm in step 3 gives $97=5\cdot 17+12$, $17=1\cdot 12+5$ and hence $b=5$ is the first number obtained in the Euclidean stage, which is less than or equal to the square root of 97. Now $c=(97-5^2)/2=36$ is a square, hence a solution (unique) to our equation is $(x,y)=(5,6)$. Of course, this could have been found much more quickly by inspection, but for larger numbers we need to use the algorithm as written.

The proof of this algorithm is not really difficult, but is a little painful so we refer to [Mor-Nic]. A nice proof due to H. W. Lenstra can be found in [Scho2]. Note also that Algorithm 1.3.14 above can also be used to solve the problem, and the proof that we gave of the validity of that algorithm is similar, but simpler.

When working in complex quadratic orders of discriminant $D<0$ congruent to 0 or 1 modulo 4 (see Chapter 5), it is more natural to solve the equation

$$x^2+|D|y^2=4p$$

where p is an odd prime (we will for example need this in Chapter 9).

If $4 \mid D$, we must have $2 \mid x$, hence the equation is equivalent to $x'^2+dy^2=p$ with $x'=x/2$ and $d=|D|/4$, which we can solve by using Algorithm 1.5.2.

If $D \equiv 1 \pmod 8$, we must have $x^2-y^2 \equiv 4 \pmod 8$ and this is possible only when x and y are even, hence our equation is equivalent to $x'^2+dy'^2=p$ with $x'=x/2$, $y'=y/2$ and $d=|D|$, which is again solved by Algorithm 1.5.2

Finally, if $D \equiv 5 \pmod 8$, the parity of x and y is not a priori determined. Therefore Algorithm 1.5.2 cannot be applied as written. There is however a modification of Algorithm 1.5.2 which enables us to treat this problem.

For this compute x_0 such that $x_0^2 \equiv D \pmod p$ using Algorithm 1.5.1, and if necessary change x_0 into $p - x_0$ so that in fact $x_0^2 \equiv D \pmod{4p}$. Then apply the algorithm as written, starting with $(a, b) = (2p, x_0)$, and stopping as soon as $b < l$, where $l = \lfloor 2\sqrt{p} \rfloor$. Then, as in [Mor-Nic] one can show that this gives the (essentially unique) solution to $x^2 + |D|y^2 = 4p$. This gives the following algorithm.

Algorithm 1.5.3 (Modified Cornacchia). Let p be a prime number and D be a negative integer such that $D \equiv 0$ or 1 modulo 4 and $|D| < 4p$. This algorithm either outputs an integer solution (x, y) to the Diophantine equation $x^2 + |D|y^2 = 4p$, or says that such a solution does not exist.

1. [Case $p = 2$] If $p = 2$ do as follows. If $D+8$ is the square of an integer, output $(\sqrt{D+8}, 1)$, otherwise say that the equation has no solution. Then terminate the algorithm.
2. [Test if residue] Using Algorithm 1.4.12 compute $k \leftarrow \left(\frac{D}{p}\right)$. If $k = -1$, say that the equation has no solution and terminate the algorithm.
3. [Compute square root] Using Shanks's Algorithm 1.5.1, compute an integer x_0 such that $x_0^2 \equiv D \bmod p$ and $0 \leq x_0 < p$, and if $x_0 \not\equiv D \pmod 2$, set $x_0 \leftarrow p - x_0$. Finally, set $a \leftarrow 2p$, $b \leftarrow x_0$ and $l \leftarrow \lfloor 2\sqrt{p} \rfloor$.
4. [Euclidean algorithm] If $b > l$, set $r \leftarrow a \bmod b$, $a \leftarrow b$, $b \leftarrow r$ and go to step 4.
5. [Test solution] If $|D|$ does not divide $4p - b^2$ or if $c = (4p - b^2)/|D|$ is not the square of an integer (see Algorithm 1.7.3), say that the equation has no solution and terminate the algorithm. Otherwise, output $(x, y) = (b, \sqrt{c})$ and terminate the algorithm.

1.6 Solving Polynomial Equations Modulo p

We will consider more generally in Chapter 3 the problem of factoring polynomials mod p. If one wants only to find the linear factors, i.e. the roots mod p, then for small degrees one can use the standard formulas. To avoid writing congruences all the time, we implicitly assume that we work in $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$.

In degree one, the solution of the equation $ax + b = 0$ is $x = -b \cdot a^{-1}$, where a^{-1} is computed using Euclid's extended algorithm.

In degree two, the solutions of the equation $ax^2 + bx + c = 0$ where $a \neq 0$ and $p \neq 2$, are given as follows. Set $D = b^2 - 4ac$. If $\left(\frac{D}{p}\right) = -1$, then there are no solutions in $\mathbb{F}_p$. If $\left(\frac{D}{p}\right) = 0$, i.e. if $p \mid D$, then there is a unique (double) solution given by $x = -b \cdot (2a)^{-1}$. Finally, if $\left(\frac{D}{p}\right) = 1$, there are two solutions,

obtained in the following way: compute an s such that $s^2 = D$ using one of the algorithms of the preceding section. Then the solutions are as usual

$$(-b \pm s) \cdot (2a)^{-1}.$$

In degree three, Cardano's formulas can be used (see Exercise 28 of Chapter 3). There are however two difficulties which must be taken care of. The first is that we must find an algorithm to compute cube roots. This can be done in a manner similar to the case of square roots. The second difficulty lies in the handling of square roots when these square roots are not in $\mathbb{F}_p$ (they are then in $\mathbb{F}_{p^2}$). This is completely analogous to handling complex numbers when a real cubic equation has three real roots. The reader will find it an amusing exercise to try and iron out all these problems (see Exercise 28). Otherwise, see [Wil-Zar] and [Mor1], who also gives the analogous recipes for degree four equations (note that for computing fourth roots one can simply compute two square roots).

In degree 5 and higher, the general equations have a non-solvable Galois group, hence as in the complex case, no special-purpose algorithms are known, and one must rely on general methods, which are slower. These methods will be seen in Section 3.4, to which we refer for notations and definitions, but in the special case of root finding, the algorithm is much simpler. We assume $p > 2$ since for $p = 2$ there are just two values to try.

Algorithm 1.6.1 (Roots Mod p). Given a prime number $p \geq 3$ and a polynomial $P \in \mathbb{F}_p[X]$, this algorithm outputs the roots of P in $\mathbb{F}_p$. This algorithm will be called recursively, and it is understood that all the operations are done in $\mathbb{F}_p$.

1. [Isolate roots in $\mathbb{F}_p$] Compute $A(X) \leftarrow (X^p - X, P(X))$ as explained below. If $A(0) = 0$, output 0 and set $A(X) \leftarrow A(X)/X$.
2. [Small degree?] If $\deg(A) = 0$, terminate the algorithm. If $\deg(A) = 1$, and $A(X) = a_1X + a_0$, output $-a_0/a_1$ and terminate the algorithm. If $\deg(A) = 2$ and $A(X) = a_2X^2 + a_1X + a_0$, set $d \leftarrow a_1^2 - 4a_0a_2$, compute $e \leftarrow \sqrt{d}$ using Algorithm 1.5.1, output $(-a_1 + e)/(2a_2)$ and $(-a_1 - e)/(2a_2)$, and terminate the algorithm. (Note that e will exist.)
3. [Random splitting] Choose a random $a \in \mathbb{F}_p$, and compute $B(X) \leftarrow ((X + a)^{(p-1)/2} - 1, A(X))$ as explained below. If $\deg(B) = 0$ or $\deg(B) = \deg(A)$, go to step 3.
4. [Recurse] Output the roots of B and A/B using the present algorithm recursively (skipping step 1), and terminate the algorithm.

Proof. The elements of $\mathbb{F}_p$ are the elements x of an algebraic closure which satisfy $x^p = x$. Hence, the polynomial A computed in step 1 is, up to a constant factor, equal to the product of the $X - x$ where the x are the roots of P in $\mathbb{F}_p$. Step 3 then splits the roots x in two parts: the roots such that $x + a$ is a quadratic residue mod p, and the others. Since a is random, this

has approximately one chance in $2^{\deg(A)-1}$ of not splitting the polynomial A into smaller pieces, and this shows that the algorithm is valid. □

Implementation Remarks.

(1) step 2 can be simplified by not taking into account the case of degree 2, but this gives a slightly less efficient algorithm. Also, if step 2 is kept as it is, it may be worthwhile to compute once and for all the quadratic non-residue mod p which is needed in Algorithm 1.5.1.

(2) When we are asked to compute a GCD of the form $\gcd(u^n - b, c)$, we must *not* compute $u^n - b$, but instead we compute $d \leftarrow u^n \bmod c$ using the powering algorithm. Then we have $\gcd(u^n - b, c) = \gcd(d - b, c)$. In addition, since $u = X + a$ is a very simple polynomial, the left-right versions of the powering algorithm (Algorithms 1.2.3 and 1.2.4) are more advantageous here.

(3) When p is small, and in particular when p is smaller than the degree of $A(X)$, it may be faster to simply test all values $X = 0, \ldots, p-1$. Thus, the above algorithm is really useful when p is not too small. In that case, it may be faster to compute $\gcd(X^{(p-1)/2} - 1, A(X - a))$ than $\gcd((X + a)^{(p-1)/2} - 1, A(X))$.

1.7 Power Detection

In many algorithms, it is necessary to detect whether a number is a square or more generally a perfect power, and if it is, to compute the root. We consider here the three most frequent problems of this sort and give simple *arithmetic* algorithms to solve them. Of course, to test whether $n = m^k$, you can always compute the nearest integer to $e^{\ln n/k}$ by transcendental means, and see if the k^{th} power of that integer is equal to n. This needs to be tried only for $k \leq \lg n$. This is clearly quite inefficient, and also requires the use of transcendental functions, so we turn to better methods.

1.7.1 Integer Square Roots

We start by giving an algorithm which computes the integer part of the square root of any positive integer n. It uses a variant of Newton's method, but works entirely with integers. The algorithm is as follows.

Algorithm 1.7.1 (Integer Square Root). Given a positive integer n, this algorithm computes the integer part of the square root of n, i.e. the number m such that $m^2 \leq n < (m+1)^2$.

1. [Initialize] Set $x \leftarrow n$ (see discussion).
2. [Newtonian step] Set $y \leftarrow \lfloor (x + \lfloor n/x \rfloor)/2 \rfloor$ using integer divides and shifts.

3. [Finished?] If $y < x$ set $x \leftarrow y$ and go to step 2. Otherwise, output x and terminate the algorithm.

Proof. By step 3, the value of x is strictly decreasing, hence the algorithm terminates. We must show that the output is correct. Let us set $q = \lfloor \sqrt{n} \rfloor$.

Since $(t + n/t)/2 \geq \sqrt{n}$ for any positive real value of t, it is clear that the inequality $x \geq q$ is satisfied throughout the algorithm (note that it is also satisfied also after the initialization step). Now assume that the termination condition in step 3 is satisfied, i.e. that $y = \lfloor (x+n/x)/2 \rfloor \geq x$. We must show that $x = q$. Assume the contrary, i.e. that $x \geq q+1$. Then,

$$y - x = \left\lfloor \frac{x + n/x}{2} \right\rfloor - x = \left\lfloor \frac{n/x - x}{2} \right\rfloor = \left\lfloor \frac{n - x^2}{2x} \right\rfloor.$$

Since $x \geq q+1 > \sqrt{n}$, we have $n - x^2 < 0$, hence $y - x < 0$ contradiction. This shows the validity of the algorithm. □

Remarks.

(1) We have written the formula in step 2 using the integer part function twice to emphasize that every operation must be done using integer arithmetic, but of course mathematically speaking, the outermost one would be enough.

(2) When actually implementing this algorithm, the initialization step must be modified. As can be seen from the proof, the only condition which must be satisfied in the initialization step is that x be greater or equal to the integer part of $\sqrt{n}$. One should try to initialize x as close as possible to this number. For example, after a $O(\ln \ln n)$ search, as in the left-right binary powering Algorithm 1.2.2, one can find e such that $2^e \leq n < 2^{e+1}$. Then, one can take $x \leftarrow 2^{\lfloor (e+2)/2 \rfloor}$. Another option is to compute a single precision floating point approximation to the square root of n and to take the ceiling of that. The choice between these options is machine dependent.

(3) Let us estimate the running time of the algorithm. As written, we will spend a lot of time essentially dividing x by 2 until we are in the right ball-park, and this requires $O(\ln n)$ steps, hence $O(\ln^3 n)$ running time. However, if care is taken in the initialization step as mentioned above, we can reduce this to the usual number of steps for a quadratically convergent algorithm, i.e. $O(\ln \ln n)$. In addition, if the precision is decreased at each iteration, it is not difficult to see that one can obtain an algorithm which runs in $O(\ln^2 n)$ bit operations, hence only a constant times slower than multiplication/division.

1.7.2 Square Detection

Given a positive integer n, we want to determine whether n is a square or not. One method of course would be to compute the integer square root of

n using Algorithm 1.7.1, and to check whether n is equal to the square of the result. This is far from being the most efficient method. We could also use Exercise 22 which says that a number is a square if and only if it is a quadratic residue modulo every prime not dividing it, and compute a few Legendre symbols using the algorithms of Section 1.4.2. We will use a variant of this method which replaces Legendre symbol computation by table lookup. One possibility is to use the following algorithm.

Precomputations 1.7.2. This is to be done and stored once and for all.

1. [Fill 11] For $k = 0$ to 10 set $q11[k] \leftarrow 0$. Then for $k = 0$ to 5 set $q11[k^2 \bmod 11] \leftarrow 1$.
2. [Fill 63] For $k = 0$ to 62 set $q63[k] \leftarrow 0$. Then for $k = 0$ to 31 set $q63[k^2 \bmod 63] \leftarrow 1$.
3. [Fill 64] For $k = 0$ to 63 set $q64[k] \leftarrow 0$. Then for $k = 0$ to 31 set $q64[k^2 \bmod 64] \leftarrow 1$.
4. [Fill 65] For $k = 0$ to 64 set $q65[k] \leftarrow 0$. Then for $k = 0$ to 32 set $q65[k^2 \bmod 65] \leftarrow 1$.

Once the precomputations are made, the algorithm is simply as follows.

Algorithm 1.7.3 (Square Test). Given a positive integer n, this algorithm determines whether n is a square or not, and if it is, outputs the square root of n. We assume that the precomputations 1.7.2 have been made.

1. [Test 64] Set $t \leftarrow n \bmod 64$ (using if possible only an and statement). If $q64[t] = 0$, n is not a square and terminate the algorithm. Otherwise, set $r \leftarrow n \bmod 45045$.
2. [Test 63] If $q63[r \bmod 63] = 0$, n is not a square and terminate the algorithm.
3. [Test 65] If $q65[r \bmod 65] = 0$, n is not a square and terminate the algorithm.
4. [Test 11] If $q11[r \bmod 11] = 0$, n is not a square and terminate the algorithm.
5. [Compute square root] Compute $q \leftarrow \lfloor \sqrt{n} \rfloor$ using Algorithm 1.7.1. If $n \neq q^2$, n is not a square and terminate the algorithm. Otherwise n is a square, output q and terminate the algorithm.

The validity of this algorithm is clear since if n is a square, it must be a square modulo k for any k. Let us explain the choice of the moduli. Note first that the number of squares modulo 64,63,65,11 is 12,16,21,6 respectively (see Exercise 23). Thus, if n is not a square, the probability that this will not have been detected in the four table lookups is equal to

$$\frac{12}{64}\frac{16}{63}\frac{21}{65}\frac{6}{11} = \frac{6}{715}$$

and this is less than one percent. Therefore, the actual computation of the integer square root in step 5 will rarely be done when n is not a square. This

is the reason for the choice of the moduli. The order in which the tests are done comes from the inequalities

$$\frac{12}{64} < \frac{16}{63} < \frac{21}{65} < \frac{6}{11}.$$

If one is not afraid to spend memory, one can also store the squares modulo $45045 = 63 \cdot 65 \cdot 11$, and then only one test is necessary instead of three, in addition to the modulo 64 test.

Of course, other choices of moduli are possible (see [Nic]), but in practice the above choice works well.

1.7.3 Prime Power Detection

The last problem we will consider in this section is that of determining whether n is a prime power or not. This is a test which is sometimes needed, for example in some of the modern factoring algorithms (see Chapter 10). We will not consider the problem of testing whether n is a power of a general number, since it is rarely needed.

The idea is to use the following proposition.

Proposition 1.7.4. *Let $n = p^k$ be a prime power. Then*

(1) *For any a we have $p \mid (a^n - a, n)$.*
(2) *If $k \geq 2$ and $p > 2$, let a be a witness to the compositeness of n given by the Rabin-Miller test 8.2.2, i.e. such that $(a, n) = 1$, and if $n - 1 = 2^t q$ with q odd, then $a^q \not\equiv 1 \pmod{n}$ and for all e such that $0 \leq e \leq t-1$ then $a^{2^e q} \not\equiv -1 \pmod{n}$. Then $(a^n - a, n)$ is a non-trivial divisor of n (i.e. is different from 1 and n).*

Proof. By Fermat's theorem, we have $a^n \equiv a \pmod{p}$, hence (1) is clear. Let us prove (2). Let a be a witness to the compositeness of n as defined above. By (1), we already know that $(a^n - a, n) > 1$. Assume that $(a^n - a, n) = n$, i.e. that $a^n \equiv a \pmod{n}$. Since $(a, n) = 1$ this is equivalent to $a^{n-1} \equiv 1 \pmod{n}$, i.e. $a^{2^t q} \equiv 1 \pmod{n}$. Let f be the smallest non-negative integer such that $a^{2^f q} \equiv 1 \pmod{n}$. Thus f exists and $f \leq t$. If we had $f = 0$, this would contradict the definition of a witness ($a^q \not\equiv 1 \pmod{n}$). So $f > 0$. But then we can write

$$p^k \mid (a^{2^{f-1} q} - 1)(a^{2^{f-1} q} + 1)$$

and since p is an odd prime, this implies that p^k divides one of the two factors. But $p^k \mid (a^{2^{f-1} q} - 1)$ contradicts the minimality of f, and $p^k \mid (a^{2^{f-1} q} + 1)$ contradicts the fact that a is a witness (we cannot have $a^{2^e q} \equiv -1 \pmod{n}$ for $e < t$), hence we have a contradiction in every case thus proving the proposition. □

This leads to the following algorithm.

Algorithm 1.7.5 (Prime Power Test). Given a positive integer $n > 1$, this algorithm tests whether or not n is of the form p^k with p prime, and if it is, outputs the prime p.

1. [Case n even] If n is even, set $p \leftarrow 2$ and go to step 4. Otherwise, set $q \leftarrow n$.
2. [Apply Rabin-Miller] By using Algorithm 8.2.2 show that either q is a probable prime or exhibit a witness a to the compositeness of q. If q is a probable prime, set $p \leftarrow q$ and go to step 4.
3. [Compute GCD] Set $d \leftarrow (a^q - a, q)$. If $d = 1$ or $d = q$, then n is not a prime power and terminate the algorithm. Otherwise set $q \leftarrow d$ and go to step 2.
4. [Final test] (Here p is a divisor of n which is almost certainly prime.) Using a primality test (see Chapters 8 and 9) prove that p is prime. If it is not (an exceedingly rare occurence), set $q \leftarrow p$ and go to step 2. Otherwise, by dividing n by p repeatedly, check whether n is a power of p or not. If it is not, n is not a prime power, otherwise output p. Terminate the algorithm.

We have been a little sloppy in this algorithm. For example in step 4, instead of repeatedly dividing by p we could use a binary search analogous to the binary powering algorithm. We leave this as an exercise for the reader (Exercise 4).

1.8 Exercises for Chapter 1

1. Write a bare-bones multi-precision package as explained in Section 1.1.2.
2. Improve your package by adding a squaring operation which operates faster than multiplication, and based on the identity $(aX + b)^2 = a^2X^2 + b^2 + ((a + b)^2 - a^2 - b^2)X$, where X is a power of the base. Test when a similar method applied to multiplication (see Section 3.1.2) becomes faster than the straightforward method.
3. Given a 32-bit non-negative integer x, assume that we want to compute quickly the highest power of 2 dividing x (32 if $x = 0$). Denoting by $e(x)$ the exponent of this power of 2, show that this can be done using the formula

$$e(x) = t[(x \hat{} (x - 1)) \bmod 37]$$

where t is a suitable table of 37 values indexed from 0 to 36, and $a \hat{} b$ denotes bitwise exclusive or (addition modulo 2 on bits). Show also that 37 is the least integer having this property, and find an analogous formula for 64-bit numbers.

4. Given two integers n and p, give an algorithm which uses ideas similar to the binary powering algorithm, to check whether n is a power of p. Also, if p is known to be prime, show that one can use only repeated squarings followed by a final divisibility test.

5. Write a version of the binary GCD algorithm which uses ideas of Lehmer's algorithm, in particular keeping information about the low order words and the high order words. Try also to write an extended version.
6. Write an algorithm which computes (u, v, d) as in Algorithm 1.3.6, by storing the partial quotients and climbing back. Compare the speed with the algorithms of the text.
7. Prove that at the end of Algorithm 1.3.6, one has $v_1 = \pm b/d$ and $v_2 = \mp a/d$, and determine the sign as a function of the number of Euclidean steps.
8. Write an algorithm for finding a solution to the system of congruences $x \equiv x_1 \pmod{m_1}$ and $x \equiv x_2 \pmod{m_2}$ assuming that $x_1 \equiv x_2 \pmod{\gcd(m_1, m_2)}$.
9. Generalizing Exercise 8 and Algorithm 1.3.12, write a general algorithm for finding an x satisfying Theorem 1.3.9.
10. Show that the use of Gauss's Algorithm 1.3.14 leads to a slightly different algorithm than Cornacchia's Algorithm 1.5.2 for solving the equation $x^2 + dy^2 = p$ (consider $\mathbf{a} = (p, 0)$ and $\mathbf{b} = (x_0, \sqrt{d})$).
11. Show how to modify Lehmer's Algorithm 1.3.13 for finding the continued fraction expansion of a real number, using the ideas of Algorithm 1.3.3, so as to avoid almost all multi-precision operations.
12. Using Algorithm 1.3.13, compute at least 30 partial quotients of the continued fraction expansions of the numbers e, e^2, e^3, $e^{2/3}$ (you will need some kind of multi-precision to do this). What do you observe? Experiment with number of the form $e^{a/b}$, and try to see for which a/b one sees a pattern. Then try and prove it (this is difficult. It is advised to start by doing a good bibliographic search).
13. Prove that if $n = n_1 n_2$ with n_1 and n_2 coprime, then $(\mathbb{Z}/n\mathbb{Z})^* \simeq (\mathbb{Z}/n_1\mathbb{Z})^* \times (\mathbb{Z}/n_2\mathbb{Z})^*$. Then prove Theorem 1.4.1.
14. Show that when $a > 2$, $g = 5$ is always a generator of the cyclic subgroup of order 2^{a-2} of $(\mathbb{Z}/2^a\mathbb{Z})^*$.
15. Prove Proposition 1.4.6.
16. Give a proof of Theorem 1.4.7 (2) along the following lines (read Chapter 4 first if you are not familiar with number fields). Let p and q be distinct odd primes. Set $\zeta = e^{2i\pi/p}$, $R = \mathbb{Z}[\zeta]$ and

$$\tau(p) = \sum_{a \bmod p} \left(\frac{a}{p}\right) \zeta^a.$$

    a) Show that $\tau(p)^2 = (-1)^{(p-1)/2} p$ and that $\tau(p)$ is invertible in R/qR.
    b) Show that $\tau(p)^q \equiv \left(\frac{q}{p}\right) \tau(p) \pmod{qR}$.
    c) Prove Theorem 1.4.7 (2), and modify the above arguments so as to prove Theorem 1.4.7 (1).
17. Prove Theorem 1.4.9 and Lemma 1.4.11.
18. Let p be an odd prime and n and integer prime to p. Then multiplication by n induces a permutation γ_n of the finite set $(\mathbb{Z}/p\mathbb{Z})^*$. Show that the signature of this permutation is equal to the Legendre symbol $\left(\frac{n}{p}\right)$. Deduce from this another proof of the quadratic reciprocity law (Theorem 1.4.7).

19. Generalizing Lemma 1.4.11, show the following general reciprocity law: if a and b are non-zero and $a = 2^\alpha a_1$ (resp. $b = 2^\beta b_1$) with a_1 and b_1 odd, then

$$\left(\frac{a}{b}\right) = (-1)^{(a_1-1)(b_1-1)/4+(\mathrm{sign}(a_1)-1)(\mathrm{sign}(b_1)-1)/4}\left(\frac{b}{a}\right).$$

20. Implement the modification suggested after Algorithm 1.4.10 (i.e. taking the smallest residue in absolute value instead of the smallest non-negative one) and compare its speed with that of the unmodified algorithm.

21. Using the quadratic reciprocity law, find the number of solutions of the congruence $x^3 \equiv 1 \pmod{p}$. Deduce from this the number of *cubic* residues mod p, i.e. numbers a not divisible by p such that the congruence $x^3 \equiv a \pmod{p}$ has a solution.

22. Show that an integer n is a square if and only if $\left(\frac{n}{p}\right) = 1$ for every prime p not dividing n.

23. Given a modulus m, give an exact formula for $s(m)$, the number of squares modulo m, in other words the cardinality of the image of the squaring map from $\mathbb{Z}/m\mathbb{Z}$ into itself. Apply your formula to the special case $m = 64, 63, 65, 11$.

24. Show that the running time of Algorithm 1.4.10 modified by keeping b odd, may be exponential time for some inputs.

25. Modify Algorithm 1.5.1 so that in addition to computing x, it also computes the (even) exponent k such that $a^q z^k = 1$ in G, using the notations of the text.

26. Give an algorithm analogous to Shanks's Algorithm 1.5.1, to find the cube roots of a mod p when a is a cubic residue. It may be useful to consider separately the cases $p \equiv 2 \pmod{3}$ and $p \equiv 1 \pmod{3}$.

27. Given a prime number p and a quadratic non-residue a mod p, we can consider $K = \mathbb{F}_{p^2} = \mathbb{F}_p(\sqrt{a})$. Explain how to do the usual arithmetic operations in K. Give an algorithm for computing square roots in K, assuming that the result is in K.

28. Generalizing Exercise 27, give an algorithm for computing cube roots in $\mathbb{F}_{p^2}$, and give also an algorithm for computing roots of equations of degree 3 by Cardano's formulas (see Exercise 28 of Chapter 3).

29. Show that, as claimed in the proof of Algorithm 1.5.1, steps 3 and 4 will require in average $e^2/4$ and at most e^2 multiplications modulo p.

30. Let $m = \prod_p p^{e_p}$ be any positive integer for which we know the complete factorization into primes, and let $a \in \mathbb{Z}$.

    a) Give a necessary and sufficient condition for a to be congruent to a square modulo m, using several Legendre symbols.

    b) Give a closed formula for the number of solutions of the congruence $x^2 \equiv a \pmod{m}$.

    c) Using Shanks's Algorithm 1.5.1 as a sub-algorithm, write an algorithm for computing a solution to $x^2 \equiv a \pmod{m}$ if a solution exists (you should take care to handle separately the power of 2 dividing m).

31. Implement Algorithm 1.6.1 with and without the variant explained in Remark (3) following the algorithm, as well as the systematic trial of $X = 0, \ldots, p-1$, and compare the speed of these three algorithms for different values of p and $\deg(P)$ or $\deg(A)$.

32. By imitating Newton's method once again, design an algorithm for computing integer cube roots which works only with integers.

33. Show that, as claimed in the text, the average number of multiplications which are not squarings in the flexible left-right base 2^k algorithm is approximately $2^{k-1}+\lg|n|/(k+1)$, and that the optimal value of k is the smallest integer such that $\lg|n|\leq(k+1)(k+2)2^{k-1}$.

34. Consider the following modification to Algorithm 1.2.4.2. We choose some odd number L such that $2^{k-1}<L<2^k$ and precompute only $z, z^3, \ldots, z^L$. Show that one can write any integer N in a unique way as $N=2^{t_0}(a_0+2^{t_1}(a_1+\ldots+2^{t_e}a_e))$ with a_i odd, $a_i\leq L$, and $t_i\geq k-1$ for $i\geq 1$, but $t_i=k-1$ only if $a_i>L-2^{k-1}$. Analyze the resulting algorithm and show that, in certain cases, it is slightly faster than Algorithm 1.2.4.2.

page 11

Perhaps surprisingly, we can easily improve on Algorithm 1.2.4 by using a flexible window of size at least k bits, instead of using a window of fixed size k. Indeed, it is easy to see that any positive integer N can be written in a unique way as

$$N = 2^{t_0}(a_0 + 2^{t_1}(a_1 + \cdots + 2^{t_e}a_e))$$

where $t_i \geq k$ for $i \geq 1$ and the a_i are odd integers such that $1 \leq a_i \leq 2^k - 1$ (in Algorithm 1.2.4 we took $t_0 = 0$, $t_i = k$ for $i \geq 1$, and $0 \leq a_i \leq 2^k - 1$ odd or even).

As before, we can precompute g^3, g^5, ... , g^{2^k-1} and then compute g^N by successive squarings and multiplications by g^{a_i}. To find the a_i and t_i, we use the following immediate sub-algorithm.

Sub-Algorithm 1.2.4.1 (Flexible Base 2^k Digits). Given a positive integer N and $k \geq 1$, this sub-algorithm computes the unique integers t_i and a_i defined above. We use $[N]_{b,a}$ to denote the integer obtained by extracting bits a through b (inclusive) of N, where bit 0 is the least significant bit.

1. [Compute t_0] Let $t_0 \leftarrow v_2(N)$, $e \leftarrow 0$ and $s \leftarrow t_0$.
2. [Compute a_e] Let $a_e \leftarrow [N]_{s+k-1,s}$.
3. [Compute t_e] Set $m \leftarrow [N]_{\infty,s+k}$. If $m = 0$, terminate the sub-algorithm. Otherwise, set $e \leftarrow e+1$, $t_e \leftarrow v_2(m)+k$, $s \leftarrow s+t_e$ and go to step 2.

The flexible window algorithm is then as follows.

Algorithm 1.2.4.2 (Flexible Left-Right Base 2^k). Given $g \in G$ and $n \in \mathbb{Z}$, this algorithm computes g^n in G. We write 1 for the unit element of G.

1. [Initialize] If $n = 0$, output 1 and terminate. If $n < 0$ set $N \leftarrow -n$ and $z \leftarrow g^{-1}$. Otherwise, set $N \leftarrow n$ and $z \leftarrow g$.
2. [Compute the a_i and t_i] Using the above sub-algorithm, compute a_i, t_i and e such that $N = 2^{t_0}(a_0 + 2^{t_1}(a_1 + \cdots + 2^{t_e}a_e))$ and set $f \leftarrow e$.
3. [Precomputations] Compute and store z^3, z^5, ... , z^{2^k-1}.
4. [Loop] If $f = e$ set $y \leftarrow z^{a_f}$ otherwise set $y \leftarrow z^{a_f} \cdot y$. Then repeat t_f times $y \leftarrow y \cdot y$.
5. [Finished?] If $f = 0$, output y and terminate the algorithm. Otherwise, set $f \leftarrow f - 1$ and go to step 4.

We have used above the word "surprisingly" to describe the behavior of this algorithm. Indeed, it is not a priori clear why it should be any better than Algorithm 1.2.4. An easy analysis shows, however, that the average number of multiplications which are not squarings is now of the order of $2^{k-1} + \lg|n|/(k+1)$ (instead of $2^{k-1} + \lg|n|/k$ in Algorithm 1.2.4), see Exercise 33. The optimal value of k is the smallest integer satisfying the inequality $\lg|n| \leq (k+1)(k+2)2^{k-1}$.

In the above example where n has 100 decimal digits, the flexible base 2^5 algorithm takes on average $(3/4)332 + 16 + 332/6 \approx 320$ multiplications, another 3% improvement. In fact, using a simple modification, in certain cases we can still easily improve (very slightly) on Algorithm 1.2.4.2, see Exercise 34.

Chapter 2

Algorithms for Linear Algebra and Lattices

2.1 Introduction

In many algorithms, and in particular in number-theoretic ones, it is necessary to use algorithms to solve common problems of linear algebra. For example, solving a linear system of equations is such a problem. Apart from stability considerations, such problems and algorithms can be solved by a single algorithm independently of the base field (or more generally of the base ring if we work with modules). Those algorithms will naturally be called *linear algebra* algorithms.

On the other hand, many algorithms of the same general kind specifically deal with problems based on specific properties of the base ring. For example, if the base ring is $\mathbb{Z}$ (or more generally any Euclidean domain), and if L is a submodule of rank n of $\mathbb{Z}^n$, then $\mathbb{Z}^n/L$ is a finite Abelian group, and we may want to know its structure once a generating system of elements of L is known. This kind of problem can loosely be called an arithmetic linear algebra problem. Such problems are trivial if $\mathbb{Z}$ is replaced by a field K. (In our example we would have $L = K^n$ hence the quotient group would always be trivial.) In fact we will see that a submodule of $\mathbb{Z}^n$ is called a *lattice*, and that essentially all arithmetic linear algebra problems deal with lattices, so we will use the term *lattice algorithms* to describe the kind of algorithms that are used for solving arithmetic linear algebra problems.

This chapter is therefore divided into two parts. In the first part, we give algorithms for solving the most common linear algebra problems. It must be emphasized that the goal will be to give general algorithms valid over any field, but that in the case of *imprecise* fields such as the field of real numbers, care must be taken to insure stability. This becomes an important problem of numerical analysis, and we refer the reader to the many excellent books on the subject ([Gol-Van], [PFTV]). Apart from mentioning the difficulties, given the spirit of this book we will not dwell on this aspect of linear algebra.

In the second part, we recall the definitions and properties of lattices. We will assume that the base ring is $\mathbb{Z}$, but essentially everything carries over to the case where the base ring is a principal ideal domain (PID), for example $K[X]$, where K is a field. Then we describe algorithms for lattices. In particular we discuss in great detail the LLL algorithm which is of fundamental importance, and give a number of applications.

2.2 Linear Algebra Algorithms on Square Matrices

2.2.1 Generalities on Linear Algebra Algorithms

Let K be a field. Linear algebra over K is the study of K-vector spaces and K-linear maps between them. We will always assume that the vector spaces that we use are finite-dimensional. Of course, infinite-dimensional vector spaces arise naturally, for example the space $K[X]$ of polynomials in one variable over K. Usually, however when one needs to perform linear algebra on these spaces it is almost always on finite-dimensional subspaces.

A K-vector space V is an abstract object, but in practice, we will assume that V is given by a basis of n linearly independent vectors $v_1, \ldots v_n$ in some K^m (where m is greater or equal, but not necessarily equal to n). This is of course highly non-canonical, but we can always reduce to that situation.

Since K^m has by definition a canonical basis, we can consider V as being given by an $m \times n$ matrix $M(V)$ (i.e. a matrix with m rows and n columns) such that the *columns* of $M(V)$ represent the coordinates in the canonical basis of K^m of the vectors v_i. If $n = m$, the linear independence of the v_i means, of course, that $M(V)$ is an invertible matrix. (The notation $M(V)$ is slightly improper since $M(V)$ is attached, not to the vector space V, but to the chosen basis v_i.)

Note that changing bases in V is equivalent to multiplying $M(V)$ on the right by an invertible $n \times n$ matrix. In particular, we may want the matrix $M(V)$ to satisfy certain properties, for example being in upper triangular form. We will see below (Algorithm 2.3.11) how to do this.

A linear map f between two vector spaces V and W of respective dimensions n and m will in practice be represented by an $m \times n$ matrix $M(f)$, $M(f)$ being the matrix of the map f with respect to the bases $M(V)$ and $M(W)$ of V and W respectively. In other words, the j-th column of $M(f)$ represents the coordinates of $f(v_j)$ in the basis w_i, where the v_j correspond to the columns of $M(V)$, and the w_i to the columns of $M(W)$.

Note that in the above we use column-representation of vectors and not row-representation; this is quite arbitrary, but corresponds to traditional usage. Once a choice is made however, one must consistently stick with it.

Thus, the objects with which we will have to work with in performing linear algebra operations are matrices and (row or column) vectors. This is only for practical purposes, but keep in mind that it rarely corresponds to anything canonical. The internal representation of vectors is completely straightforward (i.e. as a linear array).

For matrices, essentially three equivalent kinds of representation are possible. The particular one which should be chosen depends on the language in which the algorithms will be implemented. For example, it will not be the same in Fortran and in C.

One representation is to consider matrices as (row) vectors of (column) vectors. (We could also consider them as column vectors of row vectors but

the former is preferable since we have chosen to represent vectors mainly in column-representation.) A second method is to represent matrices as two-dimensional arrays. Finally, we can also represent matrices as one-dimensional arrays, by adding suitable macro-definitions so as to be able to access individual elements by row and column indices.

Whatever representation is chosen, we must also choose the index numbering for rows and columns. Although many languages such as C take 0 as the starting index, for consistency with usual mathematical notation we will assume that the first index for vectors or for rows and columns of matrices is always taken to be equal to 1. This is *not* meant to suggest that one should use this in a particular implementation, it is simply for elegance of exposition. In any given implementation, it may be preferable to make the necessary trivial changes so as to use 0 as the starting index. Again, this is a language-dependent issue.

2.2.2 Gaussian Elimination and Solving Linear Systems

The basic operation which is used in linear algebra algorithms is that of *Gaussian elimination*, sometimes also known as *Gaussian pivoting*. This consists in replacing a column (resp. a row) C by some linear combination of all the columns (resp. rows) where the coefficient of C must be non-zero, so that (for example) some coefficient becomes equal to zero. Another operation is that of exchanging two columns (resp. rows). Together, these two basic types of operations (which we will call *elementary operations* on columns or rows) will allow us to perform all the tasks that we will need in linear algebra. Note that they do not change the vector space spanned by the columns (resp. rows). Also, in matrix terms, performing a series of elementary operations on columns (resp. rows) is equivalent to right (resp. left) multiplication by an invertible square matrix of the appropriate size. Conversely, one can show (see Exercise 1) that an invertible square matrix is equal to a product of matrices corresponding to elementary operations.

The linear algebra algorithms that we give are simply adaptations of these basic principles to the specific problems that we must solve, but the underlying strategy is always the same, i.e. reduce a matrix to some simpler form (i.e. with many zeros at suitable places) so that the problem can be solved very simply. The proofs of the algorithms are usually completely straightforward, hence will be given only when really necessary. We will systematically use the following notation: if M is a matrix, M_j denotes its j-th *column*, M'_i its i-th row, and $m_{i,j}$ the entry at row i and column j. If B is a (column or row) vector, b_i will denote its i-th coordinate.

Perhaps the best way to see Gaussian elimination in action is in solving square linear systems of equations.

Algorithm 2.2.1 (Square Linear System). Let M be an $n \times n$ matrix and B a column vector. This algorithm either outputs a message saying that M is not

invertible, or outputs a column vector X such that $MX = B$. We use an auxiliary column vector C.

1. [Initialize] Set $j \leftarrow 0$.
2. [Finished?] Let $j \leftarrow j + 1$. If $j > n$ go to step 6.
3. [Find non-zero entry] If $m_{i,j} = 0$ for all $i \geq j$, output a message saying that M is not invertible and terminate the algorithm. Otherwise, let $i \geq j$ be some index such that $m_{i,j} \neq 0$.
4. [Swap?] If $i > j$, for $l = j, \ldots, n$ exchange $m_{i,l}$ and $m_{j,l}$, and exchange b_i and b_j.
5. [Eliminate] (Here $m_{j,j} \neq 0$.) Set $d \leftarrow m_{j,j}^{-1}$ and for all $k > j$ set $c_k \leftarrow dm_{k,j}$. Then, for all $k > j$ and $l > j$ set $m_{k,l} \leftarrow m_{k,l} - c_k m_{j,l}$. (Note that we do not need to compute this for $l = j$ since it is equal to zero.) Finally, for $k > j$ set $b_k \leftarrow b_k - c_k b_j$ and go to step 2.
6. [Solve triangular system] (Here M is an upper triangular matrix.) For $i = n, n-1, \ldots, 1$ (in that order) set $x_i \leftarrow (b_i - \sum_{i<j\leq n} m_{i,j}x_j)/m_{i,i}$, output $X = (x_i)_{1\leq i\leq n}$ and terminate the algorithm.

Note that steps 4 and 5 (the swap and elimination operations) are really row operations, but we have written them as working on entries since it is not necessary to take into account the first $j-1$ columns.

Note also in step 5 that we start by computing the inverse of $m_{j,j}$ since in fields like $\mathbb{F}_p$ division is usually much more time-consuming than multiplication.

The number of necessary multiplications/divisions in this algorithm is clearly asymptotic to $n^3/3$ in the general case. Note however that this does not represent the true complexity of the algorithm, which should be counted in bit operations. This of course depends on the base field (see Section 1.1.3). This remark also applies to all the other linear algebra algorithms given in this chapter.

Inverting a square matrix M means solving the linear systems $MX = E_i$, where the E_i are the canonical basis vectors of K^n, hence one can achieve this by successive applications of Algorithm 2.2.1. Clearly, it is a waste of time to use Gaussian elimination on the matrix for each linear system. (More generally, this is true when we must solve several linear systems with the same matrix M but different right hand sides B.) We should compute the inverse of M, and then the solution of a linear system requires only a simple matrix times vector multiplication requiring n^2 field multiplications.

To obtain the inverse of M, only a slight modification of Algorithm 2.2.1 is necessary.

Algorithm 2.2.2 (Inverse of a Matrix). Let M be an $n \times n$ matrix. This algorithm either outputs a message saying that M is not invertible, or outputs the inverse of M. We use an auxiliary column vector C and we recall that B_i' (resp. X_i') denotes the i-th row of B (resp. X).

1. [Initialize] Set $j \leftarrow 0$, $B \leftarrow I_n$, where I_n is the $n \times n$ identity matrix.
2. [Finished?] Let $j \leftarrow j + 1$. If $j > n$, go to step 6.
3. [Find non-zero entry] If $m_{i,j} = 0$ for all $i \geq j$, output a message saying that M is not invertible and terminate the algorithm. Otherwise, let $i \geq j$ be some index such that $m_{i,j} \neq 0$.
4. [Swap?] If $i > j$, for $l = j, \ldots, n$ exchange $m_{i,l}$ and $m_{j,l}$, and exchange the rows B'_i and B'_j.
5. [Eliminate] (Here $m_{j,j} \neq 0$.) Set $d \leftarrow m_{j,j}^{-1}$ and for all $k > j$ set $c_k \leftarrow dm_{k,j}$. Then for all $k > j$ and $l > j$ set $m_{k,l} \leftarrow m_{k,l} - c_k m_{j,l}$. (Note that we do not need to compute this for $l = j$ since it is equal to zero.) Finally, for all $k > j$ set $B'_k \leftarrow B'_k - c_k B'_j$ and go to step 2.
6. [Solve triangular system] (Here M is an upper triangular matrix.) For $i = n, n-1, \ldots, 1$ (in that order) set $X'_i \leftarrow (B'_i - \sum_{i<j\leq n} m_{i,j} X'_j)/m_{i,i}$, output the matrix X and terminate the algorithm.

It is easy to check that the number of multiplications/divisions needed is asymptotic to $4n^3/3$ in the general case. This is only four times longer than the number required for solving a single linear system. Thus as soon as more than four linear systems with the same matrix need to be solved, it is worthwhile to compute the inverse matrix.

Remarks.

(1) In step 1 of the algorithm, the matrix B is initialized to I_n. If instead, we initialize B to be any $n \times m$ matrix N for any m, the result is the matrix $M^{-1}N$, and this is of course faster than computing M^{-1} and then the matrix product. The case $m = 1$ is exactly Algorithm 2.2.1.

(2) Instead of explicitly computing the inverse of M, it is worthwhile for many applications to put M in *LUP form* , i.e. to find a lower triangular matrix L and an upper triangular matrix U such that $M = LUP$ for some permutation matrix P. (Recall that a *permutation matrix* is a square matrix whose elements are only 0 or 1 such that each row and column has exactly one 1.) Exercise 3 shows how this can be done. Once M is in this form, solving linear systems, inverting M, computing $\det(M)$, etc ... is much simpler (see [AHU] and [PFTV]).

2.2.3 Computing Determinants

To compute determinants, we can simply use Gaussian elimination as in Algorithm 2.2.1. Since the final matrix is triangular, the determinant is trivial to compute. This gives the following algorithm.

Algorithm 2.2.3 (Determinant, Using Ordinary Elimination). Let M be an $n \times n$ matrix. This algorithm outputs the determinant of M. We use an auxiliary column vector C.

1. [Initialize] Set $j \leftarrow 0$, $x \leftarrow 1$.
2. [Finished?] Let $j \leftarrow j+1$. If $j > n$ output x and terminate the algorithm.
3. [Find non-zero entry] If $m_{i,j} = 0$ for all $i \geq j$, output 0 and terminate the algorithm. Otherwise, let $i \geq j$ be some index such that $m_{i,j} \neq 0$.
4. [Swap?] If $i > j$, for $l = j, \ldots, n$ exchange $m_{i,l}$ and $m_{j,l}$, and set $x \leftarrow -x$.
5. [Eliminate] (Here $m_{j,j} \neq 0$.) Set $d \leftarrow m_{j,j}^{-1}$ and for all $k > j$ set $c_k \leftarrow d m_{k,j}$. Then for all $k > j$ and $l > j$ set $m_{k,l} \leftarrow m_{k,l} - c_k m_{j,l}$. (Note that we do not need to compute this for $l = j$ since it is equal to zero.) Finally, set $x \leftarrow x \cdot m_{j,j}$ and go to step 2.

The number of multiplications/divisions needed in this algorithm is clearly of the same order as Algorithm 2.2.1, i.e. asymptotic to $n^3/3$ in general.

Very often, this algorithm will be used in the case where the matrix M has entries in $\mathbb{Z}$ or some polynomial ring. In this case, the elimination step will introduce denominators, and these have a tendency to get very large. Furthermore, the coefficients of the intermediate matrices will be in $\mathbb{Q}$ (or some rational function field), and hence large GCD computations will be necessary which will slow down the algorithm even more. All this is of course valid for the other straightforward elimination algorithms that we have seen.

On the other hand, if the base field is a finite field $\mathbb{F}_q$, we do not have such problems. If the base field is inexact, like the real or complex numbers or the p-adic numbers, care must be taken for numerical stability. For example, numerical analysis books advise taking the largest non-zero entry (in absolute value) and not the first non-zero one found. We refer to [Gol-Van], [PFTV] for more details on these stability problems.

To overcome the problems that we encounter when the matrix M has integer coefficients, several methods can be used (and similarly when M has coefficients in a polynomial ring). The first method is to compute $\det(M)$ modulo sufficiently many primes (using Algorithm 2.2.3 which is efficient here), and then use the Chinese remainder Theorem 1.3.9 to obtain the exact value of $\det(M)$. This can be done as soon as we know an a priori upper bound for $|\det(M)|$. (We then simply choose sufficiently many primes p_i so that the product of the p_i is greater than twice the upper bound.) Such an upper bound is given by Hadamard's inequality which we will prove below (Corollary 2.5.5; note that this corollary is proved in the context of real matrices, i.e. Euclidean vector spaces, but its proof is identical for Hermitian vector spaces).

Proposition 2.2.4 (Hadamard's Inequality). *If $M = (m_{ij})_{1 \leq i,j \leq n}$ is a square matrix with complex coefficients, then*

$$|\det(M)| \leq \prod_{1 \leq i \leq n} \left(\sum_{1 \leq j \leq n} |m_{ij}|^2 \right)^{1/2} .$$

This method for computing determinants can be much faster than a direct computation using Algorithm 2.2.3, but will be slower when the number of primes needed for the Chinese remainder theorem is large. This happens because the size of the Hadamard bound is often far from ideal.

Another method is based on the following easily proved proposition due to Dodgson (alias Lewis Caroll), which is a special case of a general theorem due to Bareiss [Bar].

Proposition 2.2.5. *Let* $M_0 = (a_{i,j}^0)_{1\le i,j\le n}$ *be an* $n \times n$ *matrix where the coefficients are considered as independent variables. Set* $c_0 = 1$ *and for* $1 \le k < n$, *define recursively*

$$a_{i,j}^{(k)} = \frac{1}{c_{k-1}} \begin{vmatrix} a_{k,k}^{(k-1)} & a_{k,j}^{(k-1)} \\ a_{i,k}^{(k-1)} & a_{i,j}^{(k-1)} \end{vmatrix}, \quad M_k = (a_{i,j}^{(k)})_{k+1\le i,j\le n} \quad \text{and} \quad c_k = a_{k,k}^{(k-1)}.$$

Finally, let $c_n = a_{n,n}^{(n-1)}$. *Then all the divisions by* c_{k-1} *are exact; we have* $\det(M_k) = c_k^{n-k-1} \det(M_0)$, *and in particular* $\det(M_0) = c_n$.

Proof (Sketch). Going from M_{k-1} to M_k is essentially Gaussian elimination, except that the denominators are removed. This shows that

$$\det(M_k) = \frac{c_k^{n-k-1}}{c_{k-1}^{n-k}} \det(M_{k-1})$$

thus proving the formula for $\det(M_k)$ by induction.

That all the divisions by c_{k-1} are exact comes from the easily checked fact that we can explicitly write the coefficients $a_{i,j}^{(k)}$ as $(k+1)\times(k+1)$ minors of the matrix M_0 (see Exercise 5). □

We have stated this proposition with matrices having coefficients considered as independent variables. For more special rings, some c_k may vanish, in which case one must exchange rows or columns, as in Algorithm 2.2.3, and keep track of the sign changes. This leads to the following method for computing determinants.

Algorithm 2.2.6 (Determinant Using Gauss-Bareiss). Given an $n \times n$ matrix M with coefficients in an integral domain $\mathcal{R}$, this algorithm computes the determinant of M. All the intermediate results are in $\mathcal{R}$.

1. [Initialize] Set $k \leftarrow 0$, $c \leftarrow 1$, $s \leftarrow 1$.
2. [Increase k] Set $k \leftarrow k+1$. If $k = n$ output $sm_{n,n}$ and terminate the algorithm. Otherwise, set $p \leftarrow m_{k,k}$.
3. [Is $p = 0$?] If $p \neq 0$ go to step 4. Otherwise, look for the first non-zero coefficient $m_{i,k}$ in the k-th column, with $k+1 \le i \le n$. If no such coefficient

exists, output 0 and terminate the algorithm. If it does, for $j = k, \ldots, n$ exchange $m_{i,j}$ and $m_{k,j}$, then set $s \leftarrow -s$ and $p \leftarrow m_{k,k}$.

4. [Main step] (p is now non-zero.) For $i = k+1, \ldots, n$ and $j = k+1, \ldots, n$ set $t \leftarrow pm_{i,j} - m_{i,k}m_{k,j}$, then $m_{i,j} \leftarrow t/c$ where the division is exact. Then set $c \leftarrow p$ and go to step 2.

Although this algorithm is particularly well suited to the computation of determinants when the matrix M has integer (or similar type) entries, it can of course, be used in general. There is however a subtlety which must be taken into account when dealing with inexact entries.

Assume for example that the coefficients of M are polynomials with real coefficients. These in general will be imprecise. Then in step 4, the division t/c will, in general, not give a polynomial, but rather a rational function. This is because when we perform the Euclidean division of t by c, there may be a very small but non-zero remainder. In this case, when implementing the algorithm, it is essential to compute t/c using Euclidean division, and *discard* the remainder, if any.

The number of necessary multiplications/divisions in this modified algorithm is asymptotic to n^3 instead of $n^3/3$ in Algorithm 2.2.3, but using Gauss-Bareiss considerably improves on the time needed for the basic multiplications and divisions and this usually more than compensates for the factor of 3.

Finally, note that although we have explained the Gauss-Bareiss method for computing determinants, it can usually be applied to any other algorithmic problem using Gaussian elimination, where the coefficients are integers (see Exercise 6).

2.2.4 Computing the Characteristic Polynomial

Recall that if M is an $n \times n$ square matrix, the *characteristic polynomial* of M is the monic polynomial of degree n defined by

$$P(X) = \det(XI_n - M),$$

where as usual I_n is the $n \times n$ identity matrix. We want to compute the coefficients of $P(X)$. Note that the constant term of $P(X)$ is equal to $(-1)^n \det(M)$, and more generally the coefficients of $P(X)$ can be expressed as the sum of the so-called *principal minors* of M which are sub-determinants of M. To compute the coefficients of $P(X)$ in this manner is usually not the best way to proceed. (In fact the number of such minors grows exponentially with n.) In addition to the method which I have just mentioned, there are essentially four methods for computing $P(X)$.

The first method is to apply the definition directly, and to use the Gauss-Bareiss algorithm for computing $\det(XI_n - M)$, this matrix considered as having coefficients in the ring $K[X]$. Although computing in $K[X]$ is more expensive than computing in K, this method can be quite fast in some cases.

The second method is to apply Lagrange interpolation. In our special case, this gives the following formula.

$$\det(XI_n - M) = \sum_{k=0}^{n} \det(kI_n - M) \prod_{0 \le j \le n, j \ne k} \frac{(X-j)}{(k-j)}.$$

This formula is easily checked since both sides are polynomials of degree less than or equal to n which agree on the $n+1$ points $X = i$ for $0 \le i \le n$.

Hence, to compute the characteristic polynomial of M, it is enough to compute $n+1$ determinants, and this is usually faster than the first method. Since multiplication and division by small constants can be neglected in timing estimates, this method requires asymptotically $n^4/3$ multiplications/divisions when we use ordinary Gaussian elimination.

The third method is based on the computation of the *adjoint matrix* or *comatrix* of M, i.e. the matrix M^{adj} whose coefficient of row i and column j is equal to $(-1)^{i+j}$ times the sub-determinant of M obtained by removing row j and column i (note that i and j are reversed). From the expansion rule of determinants along rows or columns, it is clear that this matrix satisfies the identity

$$MM^{\mathrm{adj}} = M^{\mathrm{adj}}M = \det(M)I_n.$$

We give the method as an algorithm.

Algorithm 2.2.7 (Characteristic Polynomial and Adjoint Matrix). Given an $n \times n$ matrix M, this algorithm computes the characteristic polynomial $P(X) = \det(XI_n - M)$ of M and the adjoint matrix M^{adj} of M. We use an auxiliary matrix C and auxiliary elements a_i.

1. [Initialize] Set $i \leftarrow 0$, $C \leftarrow I_n$, $a_0 \leftarrow 1$.
2. [Finished?] Set $i \leftarrow i+1$. If $i = n$ set $a_n \leftarrow -\mathrm{Tr}(MC)/n$, output $P(X) \leftarrow \sum_{0 \le i \le n} a_i X^{n-i}$, $M^{\mathrm{adj}} \leftarrow (-1)^{n-1}C$ and terminate the algorithm.
3. [Compute next a_i and C] Set $C \leftarrow MC$, $a_i \leftarrow -\mathrm{Tr}(C)/i$, $C \leftarrow C + a_i I_n$ and go to step 2.

Before proving the validity of this algorithm, we prove a lemma.

Lemma 2.2.8. *Let M be an $n \times n$ matrix, $A(X)$ be the adjoint matrix of $XI_n - M$, and $P(X)$ the characteristic polynomial of M. We have the identity*

$$\mathrm{Tr}(A(X)) = P'(X).$$

Proof. Recall that the determinant is multilinear, hence the derivative of an $n \times n$ determinant is equal to the sum of the n determinants obtained by replacing the j-th column by its derivative, for $1 \le j \le n$. In our case, calling

E_j the columns of the identity matrix (i.e. the canonical basis of K^n), we have, after expanding the determinants along the j-th column

$$P'(X) = (\det(XI - M))' = \sum_{1 \le j \le n} A_{j,j}(X)$$

where $A_{j,j}(X)$ is the $(n-1)\times(n-1)$ sub-determinant of $XI-M$ obtaining by removing row and column j, i.e. $A_{j,j}$ is the coefficient of row and column j of the adjoint matrix $A(X)$, and this proves the lemma. □

Proof of the Algorithm. Call $A(X)$ the adjoint matrix of $XI_n - M$. We can write $A(X) = \sum_{0 \le i \le n-1} C_i X^{n-i-1}$ with constant matrices C_i. From the lemma, it follows that if $P(X) = \sum_{0 \le i \le n} a_i X^{n-i}$ we have

$$(n-i)a_i = \mathrm{Tr}(C_i).$$

On the other hand, since $P(X)I_n = (XI_n - M)A(X)$, we obtain by comparing coefficients $C_0 = I_n$ and for $i \ge 1$

$$C_i = MC_{i-1} + a_i I_n.$$

Taking traces, this gives $(n-i)a_i = \mathrm{Tr}(MC_{i-1}) + na_i$, i.e. $a_i = -\mathrm{Tr}(MC_{i-1})/i$. Finally, it is clear that $A(0) = C_{n-1}$ is the adjoint matrix of $-M$, hence $(-1)^{n-1}C_{n-1}$ is the adjoint matrix of M, thus showing the validity of the algorithm. □

The total number of operations is easily seen to be asymptotic to n^4 multiplications, and this may seem slower (by a factor of 3) than the method based on Lagrange interpolation. However, since no divisions are required the basic multiplication/division time is reduced considerably—especially when the matrix M has integral entries, and hence this algorithm is in fact *faster*. In addition, it gives for free the adjoint matrix of M (and even of $XI_n - M$ if we want it).

The fourth and last method is based on the notion of *Hessenberg form* of a matrix. We first compute a matrix H which is similar to M (i.e. is of the form PMP^{-1}), and in particular has the same characteristic polynomial as M, and which has the following form (Hessenberg form)

$$H = \begin{pmatrix} h_{1,1} & h_{1,2} & h_{1,3} & \dots & h_{1,n} \\ k_2 & h_{2,2} & h_{2,3} & \dots & h_{2,n} \\ 0 & k_3 & h_{3,3} & \dots & h_{3,n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & k_n & h_{n,n} \end{pmatrix}.$$

In this form, since we have a big triangle of zeros on the bottom left, it is not difficult to obtain a recursive relation for the characteristic polynomial of H,

hence of M. More precisely, if $p_m(X)$ is the characteristic polynomial of the sub-matrix of H formed by the first m rows and columns, we have $p_0(X) = 1$ and the recursion:

$$p_m(X) = (X - h_{m,m})p_{m-1}(X) - \sum_{i=1}^{m-1} \left(h_{i,m} \left(\prod_{j=i+1}^{m} k_j \right) p_{i-1}(X) \right).$$

This leads to the following algorithm.

Algorithm 2.2.9 (Hessenberg). Given an $n \times n$ matrix $M = (m_{i,j})$ with coefficients in a field, this algorithm computes the characteristic polynomial of M by first transforming M into a Hessenberg matrix as above.

1. [Initialize] Set $H \leftarrow M$, $m \leftarrow 2$.
2. [Search for non-zero] If all the $h_{i,m-1}$ with $i > m$ are equal to 0, go to step 4. Otherwise, let $i \geq m$ be the smallest index such that $h_{i,m-1} \neq 0$. Set $t \leftarrow h_{i,m-1}$. Then if $i > m$, for all $j \geq m-1$ exchange $h_{i,j}$ and $h_{m,j}$ and exchange column H_i with column H_m.
3. [Eliminate] For $i = m+1, \ldots, n$ do the following if $h_{i,m-1} \neq 0$: $u \leftarrow h_{i,m-1}/t$, for all $j \geq m$ set $h_{i,j} \leftarrow h_{i,j} - uh_{m,j}$, set $h_{i,m-1} \leftarrow 0$, and finally set column $H_m \leftarrow H_m + uH_i$.
4. [Hessenberg finished?] If $m < n-1$, set $m \leftarrow m+1$ and go to step 2.
5. [Initialize characteristic polynomial] Set $p_0(X) \leftarrow 1$ and $m \leftarrow 1$.
6. [Initialize computation] Set $p_m(X) \leftarrow (X - h_{m,m})p_{m-1}(X)$ and $t \leftarrow 1$.
7. [Compute p_m] For $i = 1, \ldots, m-1$ do the following: set $t \leftarrow th_{m-i+1,m-i}$, $p_m(X) \leftarrow p_m(X) - th_{m-i,m}p_{m-i-1}(X)$.
8. [Finished?] If $m < n$ set $m \leftarrow m+1$ and go to step 6. Otherwise, output $p_n(X)$ and terminate the algorithm.

This algorithm requires asymptotically only n^3 multiplications/divisions in the general case, and this is much better than the preceding algorithms when n is large. If M has integer coefficients however, the Hessenberg form as well as the intermediate results will usually be non-integral rational numbers, hence we lose all the advantage of the reduced operation count, since the time needed for the basic multiplications/divisions will be large. In that case, one should not use the Hessenberg algorithm directly. Instead, one should apply it to compute the characteristic polynomial modulo sufficiently many primes and use the Chinese remainder theorem, exactly as we did for the determinant. For this, we need bounds for the coefficients of the characteristic polynomial, analogous to the Hadamard bound. The following result, although not optimal, is easy to prove and gives a reasonably good estimate.

Proposition 2.2.10. *Let $M = (m_{i,j})$ be an $n \times n$ matrix, and write $\det(XI_n - M) = \sum_{0 \leq k \leq n} a_k X^{n-k}$ with $a_0 = 1$. Let B be an upper bound for the moduli of all the $m_{i,j}$. Then the coefficients a_k satisfy the inequality*

$$|a_k| \leq \binom{n}{k} k^{k/2} B^k.$$

Proof. As already mentioned, the coefficient a_k is up to sign equal to the sum of the $\binom{n}{k}$ principal $k \times k$ minors. By Hadamard's inequality (Proposition 2.2.4), each of these minors is bounded by $\prod(\sum |m_{ij}|^2)^{1/2}$ where the product and the sums have k terms. Hence the minors are bounded by $(kB^2)^{k/2} = k^{k/2}B^k$, and this gives the proposition. □

Remarks.

(1) The optimal form for computing the characteristic polynomial of a matrix would be triangular. This is however not possible if the eigenvalues of the matrix are not in the base field, hence the Hessenberg form can be considered as the second best choice.

(2) A problem related to computing the characteristic polynomial, is to compute the eigenvalues (and eigenvectors) of a matrix, say with real or complex coefficients. These are by definition the roots of the characteristic polynomial $P(X)$. Therefore, we could compute $P(X)$ using one of the above methods, then find the roots of $P(X)$ using algorithm 3.6.6 which we will see later, and finally apply algorithm 2.2.1 to get the eigenvectors. This is however *not* the way to proceed in general since much better methods based on iterative processes are available from numerical analysis (see [Gol-Van], [PFTV]), and we will not study this subject here.

2.3 Linear Algebra on General Matrices

2.3.1 Kernel and Image

We now come to linear algebra problems which deal with arbitrary $m \times n$ matrices M with coefficients in a field K. Recall from above that M can be viewed as giving a generating set for the subspace of K^m generated by the columns of M, or as the matrix of a linear map from an n-dimensional space to an m-dimensional space with respect to some bases. (Beware of the order of m and n.) It is usually conceptually easier to think of M in this way.

The first basic algorithm that we will need is for computing the kernel of M, i.e. a basis for the space of column vectors X such that $MX = 0$. The following algorithm is adapted from [Knu2].

Algorithm 2.3.1 (Kernel of a Matrix). Given an $m \times n$ matrix $M = (m_{i,j})$ with $1 \leq i \leq m$ and $1 \leq j \leq n$ having coefficients in a field K, this algorithm

outputs a basis of the kernel of M, i.e. of column vectors X such that $MX = 0$. We use auxiliary constants c_i $(1 \leq i \leq m)$ and d_i $(1 \leq i \leq n)$.

1. [Initialize] Set $r \leftarrow 0$, $k \leftarrow 1$ and for $i = 1, \ldots, m$, set $c_i \leftarrow 0$ (there is no need to initialize d_i).
2. [Scan column] If there does not exist a j such that $1 \leq j \leq m$ with $m_{j,k} \neq 0$ and $c_j = 0$ then set $r \leftarrow r + 1$, $d_k \leftarrow 0$ and go to step 4.
3. [Eliminate] Set $d \leftarrow -m_{j,k}^{-1}$, $m_{j,k} \leftarrow -1$ and for $s = k+1, \ldots, n$ set $m_{j,s} \leftarrow dm_{j,s}$. Then for all i such that $1 \leq i \leq m$ and $i \neq j$ set $d \leftarrow m_{i,k}$, $m_{i,k} \leftarrow 0$ and for $s = k+1, \ldots, n$ set $m_{i,s} \leftarrow m_{i,s} + dm_{j,s}$. Finally, set $c_j \leftarrow k$ and $d_k \leftarrow j$.
4. [Finished?] If $k < n$ set $k \leftarrow k + 1$ and go to step 2.
5. [Output kernel] (Here r is the dimension of the kernel.) For every k such that $1 \leq k \leq n$ and $d_k = 0$ (there will be exactly r such k), output the column vector $X = (x_i)_{1 \leq i \leq n}$ defined by

$$x_i = \begin{cases} m_{d_i,k}, & \text{if } d_i > 0 \\ 1, & \text{if } i = k \\ 0, & \text{otherwise.} \end{cases}$$

These r vectors form a basis for the kernel of M. Terminate the algorithm.

The proof of the validity of this algorithm is not difficult and is left as an exercise for the reader (see Exercise 8). In fact, the main point is that $c_j > 0$ if and only if $m_{j,c_j} = -1$ and all other entries in column c_j are equal to zero.

Note also that step 3 looks complicated because I wanted to give as efficient an algorithm as possible, but in fact it corresponds to elementary row operations.

Only a slight modification of this algorithm gives the image of M, i.e. a basis for the vector space spanned by the columns of M. In fact, apart from the need to make a copy of the initial matrix M, only step 5 needs to be changed.

Algorithm 2.3.2 (Image of a Matrix). Given an $m \times n$ matrix $M = (m_{i,j})$ with $1 \leq i \leq m$ and $1 \leq j \leq n$ having coefficients in a field K, this algorithm outputs a basis of the image of M, i.e. the vector space spanned by the columns of M. We use auxiliary constants c_i $(1 \leq i \leq m)$.

1. [Initialize] Set $r \leftarrow 0$, $k \leftarrow 1$ and for $i = 1, \ldots, m$, set $c_i \leftarrow 0$, and let $N \leftarrow M$ (we need to keep a copy of the initial matrix M).
2. [Scan column] If there does not exists a j such that $1 \leq j \leq m$ with $m_{j,k} \neq 0$ and $c_j = 0$ then set $r \leftarrow r + 1$, $d_k \leftarrow 0$ and go to step 4.
3. [Eliminate] Set $d \leftarrow -m_{j,k}^{-1}$, $m_{j,k} \leftarrow -1$ and for $s = k+1, \ldots, n$ set $m_{j,s} \leftarrow dm_{j,s}$. Then for all i such that $1 \leq i \leq m$ and $i \neq j$ set $d \leftarrow m_{i,k}$, $m_{i,k} \leftarrow 0$

and for $s = k+1, \ldots, n$ set $m_{i,s} \leftarrow m_{i,s} + dm_{j,s}$. Finally, set $c_j \leftarrow k$ and $d_k \leftarrow j$.

4. [Finished?] If $k < n$ set $k \leftarrow k+1$ and go to step 2.
5. [Output image] (Here $n-r$ is the dimension of the image, i.e. the rank of the matrix M.) For every j such that $1 \leq j \leq m$ and $c_j \neq 0$ (there will be exactly $n-r$ such j), output the column vector N_{c_j} (where N_k is the k-th column of the initial matrix M). These $n-r$ vectors form a basis for the image of M. Terminate the algorithm.

One checks easily that both the kernel and image algorithms require asymptotically $n^2m/2$ multiplications/divisions in general.

There are many possible variations on this algorithm for determining the image. For example if only the rank of the matrix M is needed and not an actual basis of the image, simply output the number $n-r$ in step 5. If one needs to also know the precise rows and columns that must be extracted from the matrix M to obtain a non-zero $(n-r)\times(n-r)$ determinant, we output the pairs (j, c_j) for each $j \leq m$ such that $c_j \neq 0$, where j gives the row number, and c_j the column number.

Finally, if the columns of M represent a generating set for a subspace of K^m, the image algorithm enables us to extract a basis for this subspace.

Remark. We recall the following definition.

Definition 2.3.3. *We will say that an $m \times n$ matrix M is in* column echelon form *if there exists $r \leq n$ and a strictly increasing map f from $[r+1, n]$ to $[1, m]$ satisfying the following properties.*

(1) *For $r+1 \leq j \leq n$, $m_{f(j),j} = 1$, $m_{i,j} = 0$ if $i > f(j)$ and $m_{f(k),j} = 0$ if $k < j$.*
(2) *The first r columns of M are equal to 0.*

It is clear that the definition implies that the last $n-r$ columns (i.e. the non-zero columns) of M are linearly independent.

It can be seen that Algorithm 2.3.1 gives the basis of the kernel in column echelon form. This property can be useful in other contexts, and hence, if necessary, we may assume that the basis which is output has this property. In fact we will see later that any subspace can be represented by a matrix in column echelon form (Algorithm 2.3.11).

For the image, the basis is simply extracted from the columns of M, no linear combination being taken.

2.3.2 Inverse Image and Supplement

A common problem is to solve linear systems whose matrix is either not square or not invertible. In other words, we want to generalize algorithm 2.2.1 for solving $MX = B$ where M is an $m \times n$ matrix. If X_0 is a particular solution of this system, the general solution is given by $X = X_0 + Y$ where $Y \in \ker(M)$, and $\ker(M)$ can be computed using Algorithm 2.3.1, so the only problem is to find one particular solution to our system (or to show that none exist). We will naturally call this the *inverse image* problem.

If we want the complete inverse image and not just a single solution, the best way is probably to use the kernel Algorithm 2.3.1. Indeed, consider the augmented $m \times (n+1)$ matrix M_1 obtained by adding B as an $n+1$-st column to the matrix M. If X is a solution to $MX = B$, and if X_1 is the $n+1$-vector obtained from X by adding -1 as $n+1$-st component, we clearly have $M_1 X_1 = 0$. Conversely, if X_1 is any solution of $M_1 X_1 = 0$, then either the $n+1$-st component of X_1 is equal to 0 (corresponding to elements of the kernel of M), or it is non-zero, and by a suitable normalization we may assume that it is equal to -1, and then the first n components give a solution to $MX = B$. This leads to the following algorithm.

Algorithm 2.3.4 (Inverse Image). Given an $m \times n$ matrix M and an m-dimensional column vector B, this algorithm outputs a solution to $MX = B$ or outputs a message saying that none exist. (The algorithm can be trivially modified to output the complete inverse image if desired.)

1. [Compute kernel] Let M_1 be the $m \times (n+1)$ matrix whose first n columns are those of M and whose $n+1$-st column is equal to B. Using Algorithm 2.3.1, compute a matrix V whose columns form a basis for the kernel of M_1. Let r be the number of columns of V.
2. [Solution exists?] If $v_{n+1,j} = 0$ for all j such that $1 \leq j \leq r$, output a message saying that the equation $MX = B$ has no solution. Otherwise, let $j \leq r$ be such that $v_{n+1,j} \neq 0$ and set $d \leftarrow -1/v_{n+1,j}$.
3. [Output solution] Let $X = (x_i)_{1 \leq i \leq n}$ be the column vector obtained by setting $x_i \leftarrow d v_{i,j}$. Output X and terminate the algorithm.

Note that as for the kernel algorithm, this requires asymptotically $n^2 m/2$ multiplications/divisions, hence is roughly three times slower than algorithm 2.2.1 when $n = m$.

If we want only one solution, or if we want several inverse images corresponding to the same matrix but different vectors, it is more efficient to directly use Gaussian elimination once again. A simple modification of Algorithm 2.2.2 does this as follows.

Algorithm 2.3.5 (Inverse Image Matrix). Let M be an $m \times n$ matrix and V be an $m \times r$ matrix, where $n \leq m$. This algorithm either outputs a message saying that some column vector of V is not in the image of M, or outputs an

$n \times r$ matrix X such that $V = MX$. We assume that the columns of M are linearly independent. We use an auxiliary column vector C and we recall that B'_i (resp. M'_i, X'_i) denotes the i-th row of B (resp. M, X).

1. [Initialize] Set $j \leftarrow 0$ and $B \leftarrow V$.
2. [Finished?] Let $j \leftarrow j + 1$. If $j > n$ go to step 6.
3. [Find non-zero entry] If $m_{i,j} = 0$ for all i such that $m \geq i \geq j$, output a message saying that the columns of M are not linearly independent and terminate the algorithm. Otherwise, let i be some index such that $m \geq i \geq j$ and $m_{i,j} \neq 0$.
4. [Swap?] If $i > j$, for $l = j, \ldots, n$ exchange $m_{i,l}$ and $m_{j,l}$, and exchange the rows B'_i and B'_j.
5. [Eliminate] (Here $m_{j,j} \neq 0$.) Set $d \leftarrow m_{j,j}^{-1}$ and for all k such that $m \geq k > j$ set $c_k \leftarrow dm_{k,j}$. Then for all k and l such that $m \geq k > j$ and $n \geq l > j$ set $m_{k,l} \leftarrow m_{k,l} - c_k m_{j,l}$. Finally, for all k such that $m \geq k > j$ set $B'_k \leftarrow B'_k - c_k B'_j$ and go to step 2.
6. [Solve triangular system] (Here the first n rows of M form an upper triangular matrix.) For $i = n, n-1, \ldots, 1$ (in that order) set $X'_i \leftarrow (B'_i - \sum_{i<j\leq n} m_{i,j} X'_j)/m_{i,i}$.
7. [Check rest of matrix] Check whether for each k such that $m \geq k > n$ we have $B'_k = M'_k X$. If this is not the case, output a message that some column vector of V is not in the image of M. Otherwise, output the matrix X and terminate the algorithm.

Note that in practice the columns of M represent a basis of some vector space hence are linearly independent. However, it is not difficult to modify this algorithm to work without the assumption that the columns of M are linearly independent.

Another problem which often arises is to find a *supplement* to a subspace in a vector space. The subspace can be considered as given by the coordinates of a basis on some basis of the full space, hence as an $n \times k$ matrix M with $k \leq n$ of rank equal to k. The problem is to supplement this basis, i.e. to find an *invertible* $n \times n$ matrix B such that the first k columns of B form the matrix M. A basis for a supplement of our subspace is then given by the last $n - k$ columns of B.

This can be done using the following algorithm.

Algorithm 2.3.6 (Supplement a Basis). Given an $n \times k$ matrix M with $k \leq n$ having coefficients in a field K, this algorithm either outputs a message saying that M is of rank less than k, or outputs an invertible $n \times n$ matrix B such that the first k columns of B form the matrix M. Recall that we denote by B_j the columns of B.

1. [Initialize] Set $s \leftarrow 0$ and $B \leftarrow I_n$.

2. [Finished?] If $s = k$, then output B and terminate the algorithm.
3. [Search for non-zero] Set $s \leftarrow s + 1$. Let t be the smallest $j \geq s$ such that $m_{t,s} \neq 0$, and set $d \leftarrow m_{t,s}^{-1}$. If such a $t \leq n$ does not exist, output a message saying that the matrix M is of rank less than k and terminate the algorithm.
4. [Modify basis and eliminate] Set $B_t \leftarrow B_s$ (if $t \neq s$), then set $B_s \leftarrow M_s$. Then for $j = s+1, \ldots, k$, do as follows. Exchange $m_{s,j}$ and $m_{t,j}$ (if $t \neq s$). Set $m_{s,j} \leftarrow dm_{s,j}$. Then, for all $i \neq s$ and $i \neq t$, set $m_{i,j} \leftarrow m_{i,j} - m_{i,s}m_{s,j}$. Finally, go to step 2.

Proof. This is an easy exercise in linear algebra and is left to the reader (Exercise 9). Note that the elimination part of step 4 ensures that the matrix BM stays constant throughout the algorithm, and at the end of the algorithm the first k rows of the matrix M form the identity matrix I_k, and the last $n-k$ rows are equal to 0. □

Often one needs to find the supplement of a subspace in another subspace and not in the whole space. In this case, the simplest solution is to use a combination of Algorithms 2.3.5 and 2.3.6 as follows.

Algorithm 2.3.7 (Supplement a Subspace in Another). Let V (resp. M) be an $m \times r$ (resp. $m \times n$) matrix whose columns form a basis of some subspace F (resp. E) of K^m with $r \leq n \leq m$. This algorithm either finds a basis for a supplement of F in E or outputs a message saying that F is not a subspace of E.

1. [Find new coordinates] Using Algorithm 2.3.5, find an $n \times r$ inverse image matrix X such that $V = MX$. If such a matrix does not exist, output a message saying that F is not a subspace of E and terminate the algorithm.
2. [Supplement X] Apply Algorithm 2.3.6 to the matrix X, thus giving an $n \times n$ matrix B whose first r columns form the matrix X.
3. [Supplement F in E] Let C be the $n \times n - r$ matrix formed by the last $n - r$ columns of B. Output MC and terminate the algorithm (the columns of MC will form a basis for a supplement of F in E).

Note that in addition to the error message of step 1, Algorithms 2.3.5 and 2.3.6 will also output error messages if the columns of V or M are not linearly independent.

2.3.3 Operations on Subspaces

The final algorithms that we will study concern the sum and intersection of two subspaces. If M and M' are $m \times n$ and $m \times n'$ matrices respectively, the columns of M (resp. M') span subspaces V (resp. V') of K^m. To obtain a basis for the sum $V + V'$ is very easy.

Algorithm 2.3.8 (Sum of Subspaces). Given an $m \times n$ (resp. $m \times n'$) matrix M (resp. M') whose columns span a subspace V (resp. V') of K^m, this algorithm finds a matrix N whose columns form a basis for $V + V'$.

1. [Concatenate] Let M_1 be the $m \times (n + n')$ matrix obtained by concatenating side by side the matrices M and M'. (Hence the first n columns of M_1 are those of M, the last n' those of M'.)
2. Using Algorithm 2.3.2 output a basis of the image of M_1 and terminate the algorithm.

Obtaining a basis for the intersection $V \cap V'$ is not much more difficult.

Algorithm 2.3.9 (Intersection of Subspaces). Given an $m \times n$ (resp. $m \times n'$) matrix M (resp. M') whose columns span a subspace V (resp. V') of K^m, this algorithm finds a matrix N whose columns form a basis for $V \cap V'$.

1. [Compute kernel] Let M_1 be the $m \times (n+n')$ matrix obtained by concatenating side by side the matrices M and M'. (Hence the first n columns of M_1 are those of M, the last n' those of M'.) Using Algorithm 2.3.1 compute a basis of the kernel of M_1, given by an $(n + n') \times p$ matrix N for some p.
2. [Compute intersection] Let N_1 be the $n \times p$ matrix obtained by extracting from N the first n *rows*. Set $M_2 \leftarrow MN_1$, output the matrix obtained by applying Algorithm 2.3.2 to M_2 and terminate the algorithm. (Note that if we know beforehand that the columns of M (resp. M') are also linearly independent, i.e. form a basis of V (resp. V'), we can simply output the matrix M_2 without applying Algorithm 2.3.2.)

Proof. We will constantly use the trivial fact that a column vector B is in the span of the columns of a matrix M if and only if there exists a column vector X such that $B = MX$.

Let N_1' be the $n' \times p$ matrix obtained by extracting from N the last n' rows. By block matrix multiplication, we have $MN_1 + M'N_1' = 0$. If B_i is the i-th column of $M_2 = MN_1$ then $B_i \in V$, but B_i is also equal to the opposite of the i-th column of $M'N_1'$, hence $B_i \in V'$. Conversely, let $B \in V \cap V'$. Then we can write $B = MX = M'X'$ for some column vectors X and X'. If Y is the $n+n'$-dimensional column vector whose first n (resp. last n') components are X (resp. $-X'$), we clearly have $M_1Y = 0$, hence $Y = NC$ for some column vector C. In particular, $X = N_1C$ hence $B = MN_1C = M_2C$, so B belongs to the space spanned by the columns of M_2. It follows that this space is equal to $V \cap V'$, and the image algorithm gives us a basis.

If the columns of M (resp. M') are linearly independent, then it is left as an easy exercise for the reader to check that the columns of M_2 are also linearly independent (Exercise 12), thus proving the validity of the algorithm. □

As mentioned earlier, a subspace V of K^m can be represented as an $m \times n$ matrix $M = M(V)$ whose columns are the coordinates of a basis of V on the

canonical basis of K^m. This representation depends entirely on the basis, so we may hope to find a more canonical representation. For example, how do we decide whether two subspaces V and W of K^m are equal? One method is of course to check whether every basis element of W is in the image of the matrix V and conversely, using Algorithm 2.3.4.

A better method is to represent V by a matrix having a special form, in the present case in column echelon form (see Definition 2.3.3).

Proposition 2.3.10. *If V is a subspace of K^m, there exists a unique basis of V such that the corresponding matrix $M(V)$ is in column echelon form.*

Proof. This will follow immediately from the following algorithm. □

Algorithm 2.3.11 (Column Echelon Form). Given an $m \times n$ matrix M this algorithm outputs a matrix N in column echelon form whose image is equal to the image of M (i.e. $N = MP$ for some invertible $n \times n$ matrix P).

1. [Initialize] Set $i \leftarrow m$ and $k \leftarrow n$.
2. [Search for non-zero] Search for the largest integer $j \leq k$ such that $m_{i,j} \neq 0$. If such a j does not exist, go to step 4. Otherwise, set $d \leftarrow 1/m_{i,j}$, then for $l = 1, \ldots, i$ set $t \leftarrow dm_{l,j}$, $m_{l,j} \leftarrow m_{l,k}$ (if $j \neq k$) and $m_{l,k} \leftarrow t$.
3. [Eliminate] For all j such that $1 \leq j \leq n$ and $j \neq k$ and for all l such that $1 \leq l \leq i$ set $m_{l,j} \leftarrow m_{l,j} - m_{l,k}m_{i,j}$. Finally, set $k \leftarrow k - 1$.
4. [Next row] If $i = 1$ output M and terminate the algorithm. Otherwise, set $i \leftarrow i - 1$ and go to step 2.

The proof of the validity of this algorithm is easy and left to the reader (see Exercise 11). The number of required multiplications/divisions is asymptotically $n^2(2m - n)/2$ if $n \leq m$ and $nm^2/2$ if $n > m$.

Since the non-zero columns of a matrix which is in column echelon form are linearly independent, this algorithm gives us an alternate way to compute the image of a matrix. Instead of obtaining a basis of the image as a subset of the columns, we obtain a matrix in column echelon form. This is preferable in many situations. Comparing the number of multiplications/divisions needed, this algorithm is slower than Algorithm 2.3.2 for $n \leq m$, but faster when $n > m$.

2.3.4 Remarks on Modules

We can study most of the above linear algebra problems in the context of modules over a commutative ring with unit R instead of vector spaces over a field. If the ring R is an integral domain, we can work over its field of fractions K. (This is what we did in the algorithms given above when we assumed that the matrices had integral entries.) However, this is not completely satisfactory, since the answer that we want may be different. For example, to compute the

kernel of a map defined between two free modules of finite rank (given as usual by a matrix), finding the kernel as a K-vector space is not sufficient, since we want it as an R-module. In fact, this kernel will usually not be a free module, hence cannot be represented by a matrix whose columns form a basis. One important special case where it will be free is when R is a principal ideal domain (PID, see Chapter 4). In this case all submodules of a free module of finite rank are free of finite rank. This happens when $R = \mathbb{Z}$ or $R = k[X]$ for a field k. In this case, asking for a basis of the kernel makes perfectly good sense, and the algorithm that we have given is not sufficient. We will see later (Algorithm 2.4.10) how to solve this problem.

A second difficulty arises when R is not an integral domain, because of the presence of zero-divisors. Since almost all linear algebra algorithms involve elimination, i.e. division by an element of R, we are bound at some point to get a non-zero non-invertible entry as divisor. In this case, we are in more trouble. Sometimes however, we can work around this difficulty. Let us consider for example the problem of solving a square linear system over $\mathbb{Z}/r\mathbb{Z}$, where r is not necessarily a prime. If we know the factorization of r into prime powers, we can use the Chinese remainder Theorem 1.3.9 to reduce to the case where r is a prime power. If r is prime, Algorithm 2.2.1 solves the problem, and if r is a higher power of a prime, we can still use Algorithm 2.2.1 applied to the field $K = \mathbb{Q}_p$ of p-adic numbers (see Exercise 2).

But what are we to do if we do not know the complete factorization of r? This is quite common, since as we will see in Chapters 8, 9 and 10 large numbers (say more than 80 decimal digits) are quite hard to factor. Fortunately, we do not really care. After extracting the known factors of r, we are left with a linear system modulo a new r for which we know (or expect) that it does not have any small factors (say none less than 10^6). We then simply apply Algorithm 2.2.1. Two things may happen. Either the algorithm goes through with no problem, and this will happen as long as all the elements which are used to perform the elimination (which we will call the pivots) are coprime to r. This will almost always be the case since r has no small factors. We then get the solution to the system. Note that this solution must be unique since the determinant of M, which is essentially equal to the product of the pivots, is coprime to r.

The other possibility is that we obtain a pivot p which is not coprime to r. Since the pivot is non-zero (modulo r), this means that the GCD (p, r) gives a non-trivial factor of r, hence we split r as a product of smaller (coprime) numbers and apply Algorithm 2.2.1 once again. The idea of working "as if" r was a prime can be applied to many number-theoretic algorithms where the basic assumption is that $\mathbb{Z}/r\mathbb{Z}$ is a field, and usually the same procedure can be made to work. H. W. Lenstra calls the case where working this way we find a non-trivial factor of r a *side exit*. In fact, this is sometimes the main purpose of an algorithm. For example, the elliptic curve factoring algorithm (Algorithm 10.3.3) uses exactly this kind of side exit to factor r.

2.4 $\mathbb{Z}$-Modules and the Hermite and Smith Normal Forms

2.4.1 Introduction to $\mathbb{Z}$-Modules

The most common kinds of modules that one encounters in number theory, apart from vector spaces, are evidently $\mathbb{Z}$-modules, i.e. Abelian groups. The $\mathbb{Z}$-modules V that we consider will be assumed to be *finitely generated*, in other words there exists a finite set $(v_i)_{1\le i\le k}$ of elements of V such that any element of V can be expressed as a linear combination of the v_i with integral coefficients. The basic results about such $\mathbb{Z}$-modules are summarized in the following theorem, whose proof can be found in any standard text (see for example [Lang]).

Theorem 2.4.1. *Let V be a finitely generated $\mathbb{Z}$-module (i.e. Abelian group).*

(1) *If V_{tors} is the* torsion subgroup *of V, i.e. the set of elements $v \in V$ such that there exists $m \in \mathbb{Z}\setminus\{0\}$ with $mv = 0$, then V_{tors} is a finite group, and there exists a non-negative integer n and an isomorphism*

$$V \simeq V_{\text{tors}} \times \mathbb{Z}^n$$

(the number n is called the rank *of V).*

(2) *If V is a free $\mathbb{Z}$-module (i.e. if $V \simeq \mathbb{Z}^n$, or equivalently by (1) if $V_{\text{tors}} = \{0\}$), then any submodule of V is free of rank less than or equal to that of V.*

(3) *If V is a finite $\mathbb{Z}$-module (i.e. by (1) if V is of zero rank), there exists n and a submodule L of $\mathbb{Z}^n$ (which is free by (2)) such that $V \simeq \mathbb{Z}^n/L$.*

Note that (2) and (3) are easy consequences of (1) (see Exercise 13).

This theorem shows that the study of finitely generated $\mathbb{Z}$-modules splits naturally into, on the one hand the study of finite $\mathbb{Z}$-modules (which we will usually denote by the letter G for (finite Abelian) group), and on the other hand the study of free $\mathbb{Z}$-modules of finite rank (which we will usually denote by the letter L for lattice (see Section 2.5)). Furthermore, (3) shows that these notions are in some sense dual to each other, so that we can in fact study only free $\mathbb{Z}$-modules, finite $\mathbb{Z}$-modules being considered as quotients of free modules.

Studying free modules L puts us in almost the same situation as studying vector spaces. In particular, we will usually consider L to be a submodule of some $\mathbb{Z}^m$, and we will represent L as an $m \times n$ matrix M whose columns give the coordinates of a basis of L on the canonical basis of $\mathbb{Z}^m$. Such a representation is of course not unique, since it depends on the choice of a basis for L. In the case of vector spaces, one of the ways to obtain a more canonical representation was to transform the matrix M into column echelon

form. Since this involves elimination, this is not possible anymore over $\mathbb{Z}$. Nonetheless, there exists an analogous notion which is just as useful, called the *Hermite normal form* (abbreviated HNF). Another notion, called the *Smith normal form* (abbreviated SNF) allows us to represent finite $\mathbb{Z}$-modules.

2.4.2 The Hermite Normal Form

The following definition is the analog of Definition 2.3.3 for $\mathbb{Z}$-modules.

Definition 2.4.2. *We will say that an $m \times n$ matrix $M = (m_{i,j})$ with integer coefficients is in* Hermite normal form *(abbreviated HNF) if there exists $r \leq n$ and a strictly increasing map f from $[r+1, n]$ to $[1, m]$ satisfying the following properties.*

(1) *For $r+1 \leq j \leq n$, $m_{f(j),j} \geq 1$, $m_{i,j} = 0$ if $i > f(j)$ and $0 \leq m_{f(k),j} < m_{f(k),k}$ if $k < j$.*
(2) *The first r columns of M are equal to 0.*

Remark. In the important special case where $m = n$ and $f(k) = k$ (or equivalently $\det(M) \neq 0$), M is in HNF if it satisfies the following conditions.

(1) M is an upper triangular matrix, i.e. $m_{i,j} = 0$ if $i > j$.
(2) For every i, we have $m_{i,i} > 0$.
(3) For every $j > i$ we have $0 \leq m_{i,j} < m_{i,i}$.

More generally, if $n \geq m$, a matrix M in HNF has the following shape

$$\begin{pmatrix} 0 & 0 & \dots & 0 & * & * & \dots & * \\ 0 & 0 & \dots & 0 & 0 & * & \dots & * \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & \dots & 0 & * \end{pmatrix}$$

where the last m columns form a matrix in HNF.

Theorem 2.4.3. *Let A be an $m \times n$ matrix with coefficients in $\mathbb{Z}$. Then there exists a unique $m \times n$ matrix $B = (b_{i,j})$ in HNF of the form $B = AU$ with $U \in \mathrm{GL}_n(\mathbb{Z})$, where $\mathrm{GL}_n(\mathbb{Z})$ is the group of matrices with integer coefficients which are invertible, i.e. whose determinant is equal to ± 1.*

Note that although B is unique, the matrix U will *not* be unique.

The matrix W formed by the non-zero columns of B will be called the Hermite normal form of the matrix A. Note that if A is the matrix of any *generating set* of a sub-$\mathbb{Z}$-module L of $\mathbb{Z}^m$, and not only of a basis, the columns of W give the unique basis of L whose matrix is in HNF. This basis will be called the HNF basis of the $\mathbb{Z}$-module L, and the matrix W the HNF of L.

In the special case where the $\mathbb{Z}$-module L is of rank equal to m, the matrix W will be upper triangular, and will sometimes be called the upper triangular HNF of L.

We give the proof of Theorem 2.4.3 as an algorithm.

Algorithm 2.4.4 (Hermite Normal Form). Given an $m \times n$ matrix A with integer coefficients $(a_{i,j})$ this algorithm finds the Hermite normal form W of A. As usual, we write $w_{i,j}$ for the coefficients of W, A_i (resp. W_i) for the columns of A (resp. W).

1. [Initialize] Set $i \leftarrow m$, $k \leftarrow n$, $l \leftarrow 1$ if $m \leq n$, $l \leftarrow m - n + 1$ if $m > n$.
2. [Row finished?] If all the $a_{i,j}$ with $j < k$ are zero, then if $a_{i,k} < 0$ replace column A_k by $-A_k$ and go to step 5.
3. [Choose non-zero entry] Pick among the non-zero $a_{i,j}$ for $j \leq k$ one with the smallest absolute value, say a_{i,j_0}. Then if $j_0 < k$, exchange column A_k with column A_{j_0}. In addition, if $a_{i,k} < 0$ replace column A_k by $-A_k$. Set $b \leftarrow a_{i,k}$.
4. [Reduce] For $j = 1, \ldots, k-1$ do the following: set $q \leftarrow \lfloor a_{i,j}/b \rceil$, and $A_j \leftarrow A_j - qA_k$. Then go to step 2.
5. [Final reductions] Set $b \leftarrow a_{i,k}$. If $b = 0$, set $k \leftarrow k + 1$ and go to step 6. Otherwise, for $j > k$ do the following: set $q \leftarrow \lfloor a_{i,j}/b \rceil$, and $A_j \leftarrow A_j - qA_k$.
6. [Finished?] If $i = l$ then for $j = 1, \ldots, n - k + 1$ set $W_j \leftarrow A_{j+k-1}$ and terminate the algorithm. Otherwise, set $i \leftarrow i - 1$, $k \leftarrow k - 1$ and go to step 2.

This algorithm terminates since one can easily prove that $|a_{i,k}|$ is strictly decreasing each time we return to step 2 from step 4. Upon termination, it is clear that W is in Hermite normal form, and since it has been obtained from A by elementary column operations of determinant ± 1, W is the HNF of A. We leave the uniqueness statement of Theorem 2.4.3 as an exercise for the reader (Exercise 14). □

Remarks.

(1) It is easy to modify the above algorithm (as well as the subsequent ones) so as to give the lower triangular HNF of A in the case where A is of rank equal to m.
(2) If we also want the matrix $U \in \mathrm{GL}_n(\mathbb{Z})$, it is easy to add the corresponding statements (see for example Algorithm 2.4.10).

Consider the very special case $m = 1$, $n = 2$ of this algorithm. The result will be (usually) a 1×1 matrix whose unique element is equal to the GCD $(a_{1,1}, a_{1,2})$. Hence, it is conceptually easier, and usually faster, to replace in the above algorithm divisions by (extended) GCD's. We can then choose among several available methods for computing these GCD's. This gives the following algorithm.

Algorithm 2.4.5 (Hermite Normal Form). Given an $m \times n$ matrix A with integer coefficients $(a_{i,j})$ this algorithm finds the Hermite normal form W of A. We use an auxiliary column vector B.

1. [Initialize] Set $i \leftarrow m$, $j \leftarrow n$, $k \leftarrow n$, $l = 1$ if $m \leq n$, $l = m-n+1$ if $m > n$.
2. [Check zero] If $j = 1$ go to step 4. Otherwise, set $j \leftarrow j-1$, and if $a_{i,j} = 0$ go to step 2.
3. [Euclidean step] Using Euclid's extended algorithm, compute (u, v, d) such that $ua_{i,k} + va_{i,j} = d = \gcd(a_{i,k}, a_{i,j})$, with $|u|$ and $|v|$ minimal (see below). Then set $B \leftarrow uA_k + vA_j$, $A_j \leftarrow (a_{i,k}/d)A_j - (a_{i,j}/d)A_k$, $A_k \leftarrow B$, and go to step 2.
4. [Final reductions] Set $b \leftarrow a_{i,k}$. If $b < 0$ set $A_k \leftarrow -A_k$ and $b \leftarrow -b$. Now if $b = 0$, set $k \leftarrow k+1$, and if $l > 1$ and $i = l$ set $l \leftarrow l-1$, then go to step 5, otherwise for $j > k$ do the following: set $q \leftarrow \lfloor a_{i,j}/b \rfloor$, and $A_j \leftarrow A_j - qA_k$.
5. [Finished?] If $i = l$ then for $j = 1, \ldots, n-k+1$ set $W_j \leftarrow A_{j+k-1}$ and terminate the algorithm. Otherwise, set $i \leftarrow i-1$, $k \leftarrow k-1$, $j \leftarrow k$ and go to step 2.

Important Remark. In step 3, we are asked to compute (u, v, d) with $|u|$ and $|v|$ minimal. The meaning of this is as follows. We must choose among all possible (u, v), the unique pair such that

$$-\frac{|a|}{d} < v\,\mathrm{sign}(b) \leq 0 \quad \text{and} \quad 1 \leq u\,\mathrm{sign}(a) \leq \frac{|b|}{d}.$$

In fact, the condition on u is equivalent to the condition on v and that such a pair exists and is unique is an exercise left to the reader (Exercise 15). The sign conditions are not important, they could be reversed if desired, but it is essential that when $d = |a|$, i.e. when $a \mid b$, we take $v = 0$. If this condition is not obeyed, the algorithm may enter into an infinite loop. This remark applies also to all the Hermite and Smith normal form algorithms that we shall see below.

Algorithms 2.4.4 and 2.4.5 work entirely with integers, and there are no divisions except for Euclidean divisions, hence one could expect that it behaves reasonably well with respect to the size of the integers involved. Unfortunately, this is absolutely not the case, and the coefficient explosion phenomenon occurs here also, even in very reasonable situations. For example, Hafner-McCurley ([Haf-McCur2]) give an example of a 20×20 integer matrix whose coefficients are less than or equal to 10, but which needs integers of up to 1500 decimal digits in the computations of Algorithm 2.4.4 or Algorithm 2.4.5 leading to its HNF. Hence, it is necessary to improve these algorithms.

One modification of Algorithm 2.4.5 would be for a fixed row i, instead of setting equal to zero the successive $a_{i,j}$ for $j = k-1, k-2, \ldots, 1$ by doing column operations between columns i and j, to set these $a_{i,j}$ equal to zero in the same order, but now doing operations between columns k and $k-1$,

then $k-1$ and $k-2$, and so on until columns 2 and 1, and then exchanging columns 1 and k. This idea is due to Bradley [Bra].

Still another modification is the following. In Algorithm 2.4.5, we perform the column operations as follows: $(k,k-1)$, $(k,k-2)$, ..., $(k,1)$. In the modified version just mentioned, the order is $(k,k-1)$, $(k-1,k-2)$, ..., $(2,1)$, $(1,k)$. One can also for row i do as follows. Work with the pair of columns (j_1,j_2) where a_{i,j_1} and a_{i,j_2} are the largest and second largest non-zero elements of row i with $j \leq k$. Then experiments show that the coefficient explosion is considerably reduced, and actual computational experience shows that it is faster than the preceding versions. However this is still insufficient for our needs.

When $m \leq n$ and A is of rank m (in which case W is an upper triangular matrix with non-zero determinant D), an important improvement suggested by several authors (see for example [Kan-Bac]) is to work modulo a multiple of the determinant of W, or even modulo a multiple of the *exponent* of $\mathbb{Z}^m/W$. (Note that D is equal to the order of the finite $\mathbb{Z}$-module $\mathbb{Z}^m/W$; the exponent is by definition the smallest positive integer e such that $e\mathbb{Z}^m \subset W$. It divides the determinant.)

In the case where $m=n$, we have $\det(W) = \pm\det(A)$ hence the determinant can be computed before doing the reduction if needed. In the general case however one does not know $\det(W)$ in advance, but in practice, the HNF is often used for obtaining a HNF-basis for a $\mathbb{Z}$-module L in a number field (see Chapter 4), and in that case one usually knows a multiple of the determinant of L. One can modify all of the above mentioned algorithms in this way.

These modifications are based on the following additional algorithm, essentially due to Hafner and McCurley (see [Haf-McCur2]):

Algorithm 2.4.6 (HNF Modulo D). Let A be an $m \times n$ integer matrix of rank m. Let $L = (l_{i,j})_{1\leq i,j\leq m}$ be the $m \times m$ upper triangular matrix obtained from A by doing all operations modulo D in any of the above mentioned algorithms, where D is a positive multiple of the determinant of the module generated by the columns of A (or equivalently of the determinant of the HNF of A). This algorithm outputs the true upper triangular Hermite normal form $W = (w_{i,j})_{1\leq i,j\leq m}$ of A. We write W_i and L_i for the i-th columns of W and L respectively.

1. [Initialize] Set $b \leftarrow D$, $i \leftarrow m$.

2. [Euclidean step] Using a form of Euclid's extended algorithm, compute (u,v,d) such that $ul_{i,i} + vb = d = \gcd(l_{i,i}, b)$. Then set $W_i \leftarrow (uL_i \bmod b)$ (recall that $a \bmod b$ is the least non-negative residue of a modulo b). If $d = b$ (i.e. if $b \mid l_{i,i}$) set in addition $w_{i,i} \leftarrow d$ (if $d \neq b$, this will already be true, but if $d = b$ we would have $w_{i,i} = 0$ if we do not include this additional assignment).

3. [Finished?] If $i > 1$, set $b \leftarrow b/d$, $i \leftarrow i-1$ and go to step 2. Otherwise, for $i = m-1, m-2, \ldots, 1$, and for $j = i+1, \ldots, m$ set $q \leftarrow \lfloor w_{i,j}/w_{i,i} \rfloor$, $W_j \leftarrow W_j - qW_i$. Output the matrix $W = (w_{i,j})_{1\leq i,j\leq m}$ and terminate the algorithm.

We must prove that this algorithm is valid. Since step 2 is executed exactly m times, the algorithm terminates, so what we need to prove is that the matrix W that the algorithm produces is indeed the HNF of A. For any $m \times n$ matrix M of rank m, denote by $\gamma_i(M)$ the GCD of all the $i \times i$ sub-determinants obtained from the last i rows of M for $1 \leq i \leq m$. It is clear that elementary column operations like those of Algorithms 2.4.4 or 2.4.5 leave these quantities unchanged. Furthermore, reduction modulo D changes these $i \times i$ sub-determinants by multiples of D, hence does not change the GCD of $\gamma_i(M)$ with D. It is clear that $\gamma_{m-i+1}(W) = w_{i,i} \cdots w_{m,m}$ divides $\det(W)$, hence divides D. Therefore we have:

$$\begin{aligned} w_{i,i} \cdots w_{m,m} &= \gcd(D, \gamma_{m-i+1}(W)) \\ &= \gcd(D, \gamma_{m-i+1}(A)) \\ &= \gcd(D, \gamma_{m-i+1}(L)) \\ &= \gcd(D, l_{i,i} \cdots l_{m,m}). \end{aligned} \tag{1_i}$$

hence the value given by Algorithm 2.4.6 for $w_{m,m}$ is correct. Call D_i the value of b for the value i, and set $P_i = w_{i+1,i+1} \cdots w_{m,m}$. Then if we assume that the diagonal elements $w_{j,j}$ are correct for $j > i$, we have by definition $D_i = D/P_i$. Hence, if we divide equation (1_{i+1}) by P_i we obtain

$$1 = \gcd(D_i, (l_{i+1,i+1} \cdots l_{m,m})/P_i)$$

for $1 \leq i < m$. Now if we divide equation (1_i) by P_i we obtain

$$w_{i,i} = \gcd(D_i, (l_{i,i} \cdots l_{m,m})/P_i) = \gcd(D_i, l_{i,i})$$

by the preceding formula, hence the diagonal elements of the matrix W which are output by Algorithm 2.4.6 are correct. Since W is an upper triangular matrix, it follows that its determinant is equal to the determinant of the HNF of A.

To finish the proof that Algorithm 2.4.6 is valid, we will show that the columns $W_i = (uL_i \bmod D_i)$ output by the algorithm are in the $\mathbb{Z}$-module L generated by the columns of A. By the remark just made, this will show that, in fact, the W_i are a basis of L, hence that W is obtained from A by elementary transformations. Since step 3 of the algorithm finishes to transform W into a Hermite normal form, W must be equal to the HNF of A. Since

$$W_i = \sum_{1 \leq j \leq m} c_{i,j} A_j + D_i B_i$$

where the A_j are the columns of A, B_i is a (column) vector in $\mathbb{Z}^m$ whose components of index greater than i are zero, and the $c_{i,j}$ are integers, the claim concerning the W_i follows immediately from the following lemma:

Lemma 2.4.7. *With the above notations, for every i with $1 \leq i \leq m$ and any vector B whose components of index greater than i are zero, we have $D_i B \in L$.*

Proof. Consider the $i \times i$ matrix N_i formed by the first i rows and columns of the true HNF of A. We already have proved that the diagonal elements are $w_{j,j}$ as output by the algorithm. Now if one considers $\mathbb{Z}^i$ as a submodule of $\mathbb{Z}^m$ by considering the last $m-i$ components to be equal to 0, then we see that the columns of N_i (extended by $m-i$ zeros) are $\mathbb{Z}$-linear combinations of the columns A_i of A, i.e. are in L. Now $\det(N_i) = w_{1,1} \cdots w_{i,i}$ and by definition D_i is a multiple of $w_{1,1} \cdots w_{i,i}$. Hence, if L_i is the submodule of $\mathbb{Z}^i$ generated by the columns of N_i, we have on the one hand $L_i \subset \mathbb{Z}^i \cap L$, and on the other hand, since $\det(N_i) = [\mathbb{Z}^i : L_i]$, we have $\det(N_i)\mathbb{Z}^i \subset L_i$ which implies $D_i\mathbb{Z}^i \subset L$, and this is equivalent to the statement of the lemma. This concludes the proof of the validity of Algorithm 2.4.6. □

Note that if we work modulo D in Algorithm 2.4.5, the order in which the columns are treated, which is what distinguishes Algorithm 2.4.5 from its variants, is not really important. Furthermore, the proof of Algorithm 2.4.6 shows that it is not necessary to work modulo the full multiple of the determinant D in Algorithm 2.4.5, but that at row i one can work modulo D_i, which can be much smaller. Finally, note that in step 2 of Algorithm 2.4.5, if we have worked modulo D (or D_i), it may happen that $a_{i,k} = 0$. In that case, it is necessary to set $a_{i,k} \leftarrow D_i$ (or any non-zero multiple of D_i). Combining these observations leads to the following algorithm, essentially due to Domich et al. [DKT].

It should be emphasized that all reductions modulo R should be taken in the interval $]-R/2, R/2]$, and not in the interval $[0, R[$. Otherwise, small negative coefficients will become large positive ones, and this may lead to infinite loops.

Algorithm 2.4.8 (HNF Modulo D). Given an $m \times n$ matrix A with integer coefficients $(a_{i,j})$ of rank m (hence such that $n \geq m$), and a positive integer D which is known to be a multiple of the determinant of the $\mathbb{Z}$-module generated by the columns of A, this algorithm finds the Hermite normal form W of A. We use an auxiliary column vector B.

1. [Initialize] Set $i \leftarrow m$, $j \leftarrow n$, $k \leftarrow n$, $R \leftarrow D$.
2. [Check zero] If $j = 1$ go to step 4. Otherwise, set $j \leftarrow j-1$, and if $a_{i,j} = 0$ go to step 2.
3. [Euclidean step] Using Euclid's extended algorithm, compute (u, v, d) such that $ua_{i,k} + va_{i,j} = d = \gcd(a_{i,k}, a_{i,j})$, with $|u|$ and $|v|$ minimal. Then set $B \leftarrow uA_k + vA_j$, $A_j \leftarrow ((a_{i,k}/d)A_j - (a_{i,j}/d)A_k) \bmod R$, $A_k \leftarrow B \bmod R$, and go to step 2.
4. [Next row] Using Euclid's extended algorithm, find (u, v, d) such that $ua_{i,k} + vR = d = \gcd(a_{i,k}, R)$. Set $W_i \leftarrow uA_k \bmod R$ (here taken in the interval $[0, R-1]$). If $w_{i,i} = 0$ set $w_{i,i} \leftarrow R$. For $j = i+1, \ldots, m$ set $q \leftarrow \lfloor w_{i,j}/w_{i,i} \rfloor$ and $W_j \leftarrow W_j - qW_i \bmod R$. If $i=1$, output the matrix $W = (w_{i,j})_{1 \leq i,j \leq m}$ and terminate the algorithm. Otherwise, set $R \leftarrow R/d$, $i \leftarrow i-1$, $k \leftarrow k-1$, $j \leftarrow k$, and if $a_{i,k} = 0$ set $a_{i,k} \leftarrow R$. Go to step 2.

This will be our algorithm of choice for HNF reduction, at least when some D is known and A is of rank m.

Remark. It has been noted (see Remark (2) after Algorithm 2.4.4) that it is easy to add statements so as to obtain the matrix U such that $B = AU$ where B is the $n \times m$ matrix in Hermite normal form whose non-zero columns form the HNF of A. In the case of modulo D algorithms such as the one above, it seems more difficult to do so.

2.4.3 Applications of the Hermite Normal Form

In this section, we will see a few basic applications of the HNF form of a matrix representing a free $\mathbb{Z}$-module. Further applications will be seen in the context of number fields (Chapter 4).

Image of an Integer Matrix. First note that finding the HNF of a matrix using Algorithm 2.4.5 is essentially analogous to finding the column echelon form in the case of vector spaces (Algorithm 2.3.11). In particular, if the columns of the matrix represents a generating set for a free module L, Algorithm 2.4.5 allows us to find a basis (in fact of quite a special form), hence it also performs the same role as Algorithm 2.3.2. Contrary to the case of vector spaces, however, it is not possible in general to extract a basis from a generating set (this would mean that $(a, b) = |a|$ or $(a, b) = |b|$ in the case $m = 1$, $n = 2$), hence an analog of Algorithm 2.3.2 cannot exist.

Kernel of an Integer Matrix. We can also use Algorithm 2.4.5 to find the kernel of an $m \times n$ integer matrix A, i.e. a $\mathbb{Z}$-basis for the free sub-$\mathbb{Z}$-module of $\mathbb{Z}^n$ which is the set of column vectors X such that $AX = 0$. Note that this *cannot* be done (at least not without considerable extra work) by using Algorithm 2.3.1 which gives only a $\mathbb{Q}$-basis. What we must do is simply keep track of the matrix $U \in \mathrm{GL}_n(\mathbb{Z})$ such that $B = AU$ is in HNF. Indeed, we have the following proposition.

Proposition 2.4.9. *Let A be an $m \times n$ matrix, $B = AU$ its HNF with $U \in \mathrm{GL}_n(\mathbb{Z})$, and let r be such that the first r columns of B are equal to 0. Then a $\mathbb{Z}$-basis for the kernel of A is given by the first r columns of U.*

Proof. If U_i is the i-th column of U, then AU_i is the i-th column of B so is equal to 0 if $i \leq r$. Conversely, let X be a column vector such that $AX = 0$ or equivalently $BY = 0$ with $Y = U^{-1}X$. Solving the system $BY = 0$ from bottom up, $b_{f(k),k} > 0$ for $k > r$ (with the notation of Definition 2.4.2) implies that the last $n - r$ coordinates of Y are equal to 0, and the first r are arbitrary, hence the first r canonical basis elements of $\mathbb{Z}^n$ form a $\mathbb{Z}$-basis for the kernel of B, and upon left multiplication by U we obtain the proposition.

□

This gives the following algorithm.

Algorithm 2.4.10 (Kernel over $\mathbb{Z}$). Given an $m \times n$ matrix A with integer coefficients $(a_{i,j})$, this algorithm finds a $\mathbb{Z}$-basis for the kernel of A. We use an auxiliary column vector B and an auxiliary $n \times n$ matrix U.

1. [Initialize] Set $i \leftarrow m$, $j \leftarrow n$, $k \leftarrow n$, $U \leftarrow I_n$, $l \leftarrow 1$ if $m \leq n$, $l \leftarrow m-n+1$ if $m > n$.
2. [Check zero] If $j = 1$ go to step 4. Otherwise, set $j \leftarrow j-1$, and if $a_{i,j} = 0$ go to step 2.
3. [Euclidean step] Using Euclid's extended algorithm, compute (u, v, d) such that $ua_{i,k} + va_{i,j} = d = \gcd(a_{i,k}, a_{i,j})$, with $|u|$ and $|v|$ minimal. Then set $B \leftarrow uA_k + vA_j$, $A_j \leftarrow (a_{i,k}/d)A_j - (a_{i,j}/d)A_k$, $A_k \leftarrow B$; similarly set $B \leftarrow uU_k + vU_j$, $U_j \leftarrow (a_{i,k}/d)U_j - (a_{i,j}/d)U_k$, $U_k \leftarrow B$, then go to step 2.
4. [Final reductions] Set $b \leftarrow a_{i,k}$. If $b < 0$ set $A_k \leftarrow -A_k$, $U_k \leftarrow -U_k$ and $b \leftarrow -b$. Now if $b = 0$, set $k \leftarrow k+1$ and go to step 5, otherwise for $j > k$ do the following: set $q \leftarrow \lfloor a_{i,j}/b \rfloor$, $A_j \leftarrow A_j - qA_k$ and $U_j \leftarrow U_j - qU_k$.
5. [Finished?] If $i = l$ then for $j = 1, \ldots, k-1$ set $M_j \leftarrow U_j$, output the matrix M and terminate the algorithm. Otherwise, set $i \leftarrow i-1$, $k \leftarrow k-1$, $j \leftarrow k$ and go to step 2.

Remark. Although this algorithm correctly gives a $\mathbb{Z}$-basis for the kernel of A, the coefficients that are obtained are usually large. To obtain a really useful algorithm, it is necessary to *reduce* the basis that is obtained, for example using one of the variants of the LLL algorithm that we will see below (see Section 2.6). However, it is desirable to obtain directly a basis of good quality that avoids introducing large coefficients. This can be done using the MLLL algorithm (see Algorithm 2.7.2), and gives an algorithm which is usually preferable.

In view of the applications to number fields, limiting ourselves to free submodules of some $\mathbb{Z}^m$ is a little too restrictive. In what follows we will simply say that L is a *module* if it is a free sub-$\mathbb{Z}$-module of rank m of $\mathbb{Q}^m$. Considering basis elements of L, it is clear that there exists a minimal positive integer d such that $dL \subset \mathbb{Z}^m$. We will call d the *denominator* of L with respect to $\mathbb{Z}^m$. Then the HNF of L will be by definition the pair (W, d), where W is the HNF of dL, and d is the denominator of L.

Test for Equality. Since the HNF representation of a free module L is unique, it is clear that one can trivially test equality of modules: their denominator and their HNF must be the same.

Sum of Modules. Given two modules L and L' by their HNF, we can compute their sum $L + L' = \{x + x', x \in L, x' \in L'\}$ in the following way. Let (W, d) and (W', d') be their HNF representation. Let $D = dd'/(d, d')$ be the least common multiple of d and d'. Denoting as usual by A_i the i-th column

of a matrix A, consider the $m \times 2m$ matrix A such that $A_i = (D/d)W_i$ and $A_{m+i} = (D/d')W'_i$ for $1 \le i \le m$, then it is clear that the columns of A generate $D(L + L')$, hence if we compute the HNF H of A and divide D and H by the greatest common divisor of D and of all the coefficients of H, we obtain the HNF normal form of $L + L'$. Apart from the treatment of denominators, this is similar to Algorithm 2.3.8.

Test for Inclusion. To test whether $L' \subset L$, where L and L' are given by their HNF, the most efficient way is probably to compute $N = L+L'$ as above, and then test the equality $N = L$. Note that if d and d' are the denominators of L and L' respectively, a necessary condition for $L' \subset L$ is that $d' \mid d$, hence the LCM D must be equal to d.

Product by a Constant. This is especially easy: if $c = p/q \in \mathbb{Q}$ with $(p, q) = 1$ and $q > 0$, the HNF of cL is obtained as follows. Let d_1 be the GCD of all the coefficients of the HNF of L. Then the denominator of cL is $qd/((p, d)(q, d_1))$, and the HNF matrix is equal to $p/((p, d)(q, d_1))$ times the HNF matrix of L.

We will see that the HNF is quite practical for other problems also, but the above list is, I hope, sufficiently convincing.

2.4.4 The Smith Normal Form and Applications

We have seen that the Hermite normal form permits us to handle free $\mathbb{Z}$-modules of finite rank quite nicely. We would now like a similar notion which would allow us to handle finite $\mathbb{Z}$-modules G. Recall from Theorem 2.4.1 (3) that such a module is isomorphic (in many ways of course) to a quotient $\mathbb{Z}^n/L$ where L is a (necessarily free) submodule of $\mathbb{Z}^n$ of rank equal to n. More elegantly perhaps, we can say that G is isomorphic to a quotient L'/L of free $\mathbb{Z}$-modules of the same (finite) rank n. Thus we can represent G (still non-canonically) by an $n \times n$ matrix A giving the coordinates of some $\mathbb{Z}$-basis of L on some $\mathbb{Z}$-basis of L'. In particular, A will have non-zero determinant, and in fact the absolute value of the determinant of A is equal to the cardinality of G, i.e. to the index $[L' : L]$ (see Exercise 18).

The freedom we now have is as follows. Changing the $\mathbb{Z}$-basis of L is equivalent to right multiplication of A by a matrix $U \in \mathrm{GL}_n(\mathbb{Z})$, as in the HNF case. Changing the $\mathbb{Z}$-basis of L' is on the other hand equivalent to *left* multiplication of A by a matrix $V \in \mathrm{GL}_n(\mathbb{Z})$. In other words, we are allowed to perform elementary column *and* row operations on the matrix A without changing (the isomorphism class of) G. This leads to the notion of *Smith normal form* of A.

Definition 2.4.11. *We say that an $n \times n$ matrix B is in Smith normal form (abbreviated SNF) if B is a diagonal matrix with nonnegative integer coefficients such that $b_{i+1,i+1} \mid b_{i,i}$ for all $i < n$.*

Then the basic theorem which explains the use of this definition is as follows.

Theorem 2.4.12. *Let A be an $n \times n$ matrix with coefficients in $\mathbb{Z}$ and non-zero determinant. Then there exists a unique matrix in Smith normal form B such that $B = VAU$ with U and V elements of $\mathrm{GL}_n(\mathbb{Z})$.*

If we set $d_i = b_{i,i}$, the d_i are called the *elementary divisors* of the matrix A, and the theorem can be written

$$A = V^{-1} \begin{pmatrix} d_1 & 0 & \dots & 0 \\ 0 & d_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & d_n \end{pmatrix} U^{-1}$$

with $d_{i+1} \mid d_i$ for $1 \leq i < n$.

This theorem, stated for matrices, is equivalent to the following theorem for $\mathbb{Z}$-modules.

Theorem 2.4.13 (Elementary Divisor Theorem). *Let L be a $\mathbb{Z}$-submodule of a free module L' and of the same rank. Then there exist positive integers $d_1, \dots, d_n$ (called the elementary divisors of L in L') satisfying the following conditions:*

(1) *For every i such that $1 \leq i < n$ we have $d_{i+1} \mid d_i$.*
(2) *As $\mathbb{Z}$-modules, we have the isomorphism*

$$L'/L \simeq \bigoplus_{1 \leq i \leq n} (\mathbb{Z}/d_i\mathbb{Z}),$$

and in particular $[L' : L] = d_1 \cdots d_n$ and d_1 is the exponent of L'/L.
(3) *There exists a $\mathbb{Z}$-basis $(v_1, \dots, v_n)$ of L' such that $(d_1 v_1, \dots, d_n v_n)$ is a $\mathbb{Z}$-basis of L.*

Furthermore, the d_i are uniquely determined by L and L'.

Remarks.

(1) This fundamental theorem is valid more generally. It holds for finitely generated (torsion) free modules over a principal ideal domain (PID, see Chapter 4). It is *false* if the base ring R is not a PID: applying the theorem to $n = 1$, $L' = R$ and L any integral ideal of R, it is clear that the truth of this theorem is equivalent to the PID condition.
(2) We have stated Theorem 2.4.12 only for square matrices of non-zero determinant. As in the Hermite case, it would be easy to state a generalization valid for general matrices (including non-square ones). In practice, this is not really needed since we can always first perform a Hermite reduction.

The proof of these two theorems can be found in any standard textbook but it follows immediately from the algorithm below.

Since we are going to deal with square matrices, as with the case of the HNF, it is worthwhile to work modulo the determinant (or a multiple). In most cases this determinant (or a multiple of it) is known in advance. It should also be emphasized again that all reductions modulo R should be taken in the interval $]-R/2, R/2]$, and not in the interval $[0, R[$.

The following algorithm is essentially due to Hafner and McCurley (see [Haf-McCur2]).

Algorithm 2.4.14 (Smith Normal Form). Given an $n \times n$ non-singular integral matrix $A = (a_{i,j})$, this algorithm finds the Smith normal form of A, i.e. outputs the diagonal elements d_i such that $d_{i+1} \mid d_i$. Recall that we denote by A_i (resp. A'_i) the columns (resp. the rows) of the matrix A. We use a temporary (column or row) vector variable B.

1. [Initialize i] Set $i \leftarrow n$, $R \leftarrow |\det(A)|$. If $n = 1$, output $d_1 \leftarrow R$ and terminate the algorithm.
2. [Initialize j for row reduction] Set $j \leftarrow i$, $c \leftarrow 0$.
3. [Check zero] If $j = 1$ go to step 5. Otherwise, set $j \leftarrow j - 1$. If $a_{i,j} = 0$ go to step 3.
4. [Euclidean step] Using Euclid's extended algorithm, compute (u, v, d) such that $ua_{i,i} + va_{i,j} = d = \gcd(a_{i,i}, a_{i,j})$, with u and v minimal (see remark after Algorithm 2.4.5). Then set $B \leftarrow uA_i + vA_j$, $A_j \leftarrow ((a_{i,i}/d)A_j - (a_{i,j}/d)A_i) \bmod R$, $A_i \leftarrow B \bmod R$ and go to step 3.
5. [Initialize j for column reduction] Set $j \leftarrow i$.
6. [Check zero] If $j = 1$ go to step 8. Otherwise, set $j \leftarrow j - 1$, and if $a_{j,i} = 0$ go to step 6.
7. [Euclidean step] Using Euclid's extended algorithm, compute (u, v, d) such that $ua_{i,i} + va_{j,i} = d = \gcd(a_{i,i}, a_{j,i})$, with u and v minimal (see remark after Algorithm 2.4.5). Then set $B \leftarrow uA'_i + vA'_j$, $A'_j \leftarrow ((a_{i,i}/d)A'_j - (a_{j,i}/d)A'_i) \bmod R$, $A'_i \leftarrow B \bmod R$, $c \leftarrow c + 1$ and go to step 6.
8. [Repeat stage i?] If $c > 0$ go to step 2.
9. [Check the rest of the matrix] Set $b \leftarrow a_{i,i}$. For $1 \leq k, l < i$ check whether $b \mid a_{k,l}$. As soon as some coefficient $a_{k,l}$ is not divisible by b, set $A'_i \leftarrow A'_i + A'_k$ and go to step 2.
10. [Next stage] (Here all the $a_{k,l}$ for $1 \leq k, l < i$ are divisible by b). Output $d_i = \gcd(a_{i,i}, R)$ and set $R \leftarrow R/d_i$. If $i = 2$, output $d_1 = \gcd(a_{1,1}, R)$ and terminate the algorithm. Otherwise, set $i \leftarrow i - 1$ and go to step 2.

This algorithm seems complicated at first, but one can see that it is actually quite straightforward, using elementary row and column operations of determinant ± 1 to reduce the matrix A.

This algorithm terminates (and does not take too many steps!) since each time one returns to step 2 from step 9, the coefficient $a_{i,i}$ has been reduced at least by a factor of 2.

The proof that this algorithm is valid, i.e. that the result is correct, follows exactly the proof of the validity of Algorithm 2.4.6. If we never reduced modulo R in Algorithm 2.4.14, it is clear that the result would be correct (however the coefficients would explode). Incidentally, this gives a proof of Theorems 2.4.12 and 2.4.13.

Hence, we must simply show that the transformations done in step 10 correctly restore the values of d_i. Denote by $\delta_i(A)$ the GCD of the determinants of *all* $i \times i$ sub-matrices of A, and not only from the first i rows as in the proof of Algorithm 2.4.6. Then, in a similar manner, these δ_i are invariant under elementary row and column operations of determinant ± 1. Hence, denoting by Δ the diagonal SNF of A, by D the determinant of A, and by $S = (a_{i,j})$ the final form of the matrix A at the end of Algorithm 2.4.14, we have:

$$\begin{aligned} d_i \cdots d_n &= \gcd(D, \delta_{n-i+1}(\Delta)) \\ &= \gcd(D, \delta_{n-i+1}(A)) \\ &= \gcd(D, \delta_{n-i+1}(S)) \\ &= \gcd(D, a_{i,i} \cdots a_{n,n}). \end{aligned} \tag{2_i}$$

Hence, if we set $P_i = d_{i+1} \cdots d_n$, exactly as in the proof of Algorithm 2.4.6 we obtain

$$1 = (D/P_i, (a_{i+1,i+1} \cdots a_{n,n})/P_i)$$

(divide formula (2_{i+1}) by P_i), then

$$d_i = (D/P_i, (a_{i,i} a_{i+1,i+1} \cdots a_{n,n})/P_i)$$

(divide (2_i) by P_i), and hence

$$d_i = (D/P_i, a_{i,i}).$$

But clearly in stage i of the algorithm, $R = D/P_i$, thus proving the validity of the algorithm. □

Note that we have chosen an order for the d_i which is consistent with our choice for Hermite normal forms, but which is the reverse of the one which is found in most texts. The modifications to Algorithm 2.4.14 so that the order is reversed are trivial (essentially make i and j go up instead of down) and are left to the reader.

The Smith normal form will mainly be used as follows. Let G be a finite $\mathbb{Z}$-module (i.e. a finite Abelian group). We want to determine the structure of G, and in particular its cardinality. Note that a corollary of Theorem 2.4.13 is the structure theorem for finite Abelian groups: such a group is isomorphic to a unique direct sum of cyclic groups $\mathbb{Z}/d_i\mathbb{Z}$ with $d_{i+1} \mid d_i$.

We can then proceed as follows. By theoretical means, we find some integer n and a free module L' of rank n such that G is isomorphic to a quotient L'/L, where L is also of rank n but unknown. We then determine as many elements of L as possible (how to do this depends, of course, entirely on the specific problem) so as to have at least n elements which are $\mathbb{Q}$-linearly independent. Using the Hermite normal form Algorithm 2.4.5, we can then find the HNF basis for the submodule L_1 of L generated by the elements that we have found. Computing the determinant of this basis (which is trivial since the basis is in triangular form) already gives us the cardinality of L'/L_1. If we know bounds for the order of G (for example, if we know the order of G up to a factor of $\sqrt{2}$ from above and below), we can check whether $L_1 = L$. If not, we continue finding new elements of L until the cardinality check shows that $L_1 = L$. We can then compute the SNF of the HNF basis (note that the determinant is now known), and this gives us the complete structure of G.

We will see a concrete application of the process just described in the sub-exponential computations of class groups (see Chapter 5).

Remark. The diagonal elements which are obtained after a Hermite Normal Form computation are usually *not* equal to the Smith invariants. For example, the matrix $\begin{pmatrix} 2 & 1 \\ 0 & 2 \end{pmatrix}$ is in HNF, but its Smith normal form has as diagonal elements $(4, 1)$.

2.5 Generalities on Lattices

2.5.1 Lattices and Quadratic Forms

We are now going to add some extra structure to free $\mathbb{Z}$-modules of finite rank. Recall the following definition.

Definition 2.5.1. *Let K be a field of characteristic different from 2, and let V be a K-vector space. We say that a map q from V to K is a* quadratic form *if the following two conditions are satisfied:*

(1) *For every $\lambda \in K$ and $x \in V$ we have*

$$q(\lambda \cdot x) = \lambda^2 q(x).$$

(2) *If we set $b(x, y) = \frac{1}{2}(q(x+y) - q(x) - q(y))$ then b is a (symmetric) bilinear form, i.e. $b(x + x', y) = b(x, y) + b(x', y)$ and $b(\lambda \cdot x, y) = \lambda b(x, y)$ for all $\lambda \in K$, x, x' and y in V (the similar conditions on the second variable follow from the fact that $b(y, x) = b(x, y)$).*

The identity $b(x, x) = q(x)$ allows us to recover q from b.

In the case where $K = \mathbb{R}$, we say that q is *positive definite* if for all non-zero $x \in V$ we have $q(x) > 0$.

Definition 2.5.2. *A* lattice L *is a free* $\mathbb{Z}$*-module of finite rank together with a positive definite quadratic form* q *on* $L \otimes \mathbb{R}$.

Let $(\mathbf{b}_i)_{1 \le i \le n}$ be a $\mathbb{Z}$-basis of L. If $\mathbf{x} = \sum_{1 \le i \le n} x_i \mathbf{b}_i \in L$ with $x_i \in \mathbb{Z}$, the definition of a quadratic form implies that

$$q(\mathbf{x}) = \sum_{1 \le i,j \le n} q_{i,j} x_i x_j \quad \text{with } q_{i,j} = b(\mathbf{b}_i, \mathbf{b}_j)$$

where as above, b denotes the symmetric bilinear form associated to q.

The matrix $Q = (q_{i,j})_{1 \le i,j \le n}$ is then a symmetric matrix which is positive definite when q is positive definite. We have $b(\mathbf{x}, \mathbf{y}) = Y^t Q X$ and in particular $q(\mathbf{x}) = X^t Q X$ where X and Y are the column vectors giving the coordinates of $\mathbf{x}$ and $\mathbf{y}$ respectively in the basis $(\mathbf{b}_i)$.

We will say that two lattices (L, q) and (L', q') are equivalent if there exists a $\mathbb{Z}$-module isomorphism between L and L' sending q to q'. We will identify equivalent lattices. Also, when the quadratic form is understood, we will write L instead of (L, q).

A lattice (L, q) can be represented in several ways all of which are useful. First, one can choose a $\mathbb{Z}$-basis $(\mathbf{b}_i)_{1 \le i \le n}$ of the lattice. Then an element of $\mathbf{x} \in L$ will be considered as a (column) vector X giving the (integral) coordinates of $\mathbf{x}$ on the basis. The quadratic form q is then represented by the positive definite symmetric matrix Q as we have seen above.

Changing the $\mathbb{Z}$-basis amounts to replacing X by PX for some $P \in \mathrm{GL}_n(\mathbb{Z})$, hence $q(x) = (PX)^t Q (PX) = X^t Q' X$ with $Q' = P^t Q P$. Hence, equivalence classes of lattices correspond to equivalence classes of positive definite symmetric matrices under the equivalence relation $Q' \sim Q$ if and only if there exists $P \in \mathrm{GL}_n(\mathbb{Z})$ such that $Q' = P^t Q P$. Note that $\det(P) = \pm 1$, hence the determinant of Q is independent of the choice of the basis. Since Q is positive definite, $\det(Q) > 0$ and we will set $d(L) = \det(Q)^{1/2}$ and call it the *determinant of the lattice*.

A second way to represent a lattice (L, q) is to consider L as a discrete subgroup of rank n of the Euclidean vector space $E = L \otimes \mathbb{R}$. Then if $(\mathbf{b}_i)_{1 \le i \le n}$ is a $\mathbb{Z}$-basis of L, it is also by definition of the tensor product an $\mathbb{R}$-basis of E. The matrix of scalar products $Q = (\mathbf{b}_i \cdot \mathbf{b}_j)_{1 \le i,j \le n}$ (where $\mathbf{b}_i \cdot \mathbf{b}_j = b(\mathbf{b}_i, \mathbf{b}_j)$) is then called the *Gram matrix* of the $\mathbf{b}_i$. If we choose some orthonormal basis of E, we can then identify E with the Euclidean space $\mathbb{R}^n$ with the usual Euclidean structure coming from the quadratic form $q(\mathbf{x}) = x_1^2 + \cdots + x_n^2$.

If B is the $n \times n$ matrix whose columns give the coordinates of the $\mathbf{b}_i$ on the chosen orthonormal basis of E, it is clear that $Q = B^t B$. In particular, $d(L) = |\det(B)|$. Furthermore, if another choice of orthonormal basis is made, the new matrix B' will be of the form $B' = KB$ where K is an *orthogonal*

matrix, i.e. a matrix such that $K^tK = KK^t = I_n$. Thus we have proved the following proposition.

Proposition 2.5.3.

(1) *If Q is the matrix of a positive definite quadratic form, then Q is the Gram matrix of some lattice basis, i.e. there exists a matrix $B \in \mathrm{GL}_n(\mathbb{R})$ such that $Q = B^tB$*

(2) *The Gram matrix of a lattice basis $\mathbf{b}_i$ determines this basis uniquely up to isometry. In other words, if the $\mathbf{b}_i$ and the $\mathbf{b}'_i$ have the same Gram matrix, then the $\mathbf{b}'_i$ can be obtained from the $\mathbf{b}_i$ by an orthogonal transformation. In matrix terms, $B' = KB$ where K is an orthogonal matrix.*

It is not difficult to give a completely matrix-theoretic proof of this proposition (see Exercise 20).

It follows from the above results that when dealing with lattices, it is not necessary to give the coordinates of the $\mathbf{b}_i$ on some orthonormal basis. We can simply give a positive definite matrix which we can then think of as being the Gram matrix of the $\mathbf{b}_i$.

We see from the above discussion that there are natural bijections between the following three sets.

$$\{\text{Isomorphism classes of lattices of rank } n\},$$

$$\{\text{Classes of positive definite symmetric matrices } Q\}/\sim,$$

where $Q' \sim Q$ if and only if $Q' = P^tQP$ for some $P \in \mathrm{GL}_n(\mathbb{Z})$, and

$$\mathrm{GL}_n(\mathbb{R})/\sim,$$

where $B' \sim B$ if and only if $B' = KBP$ for some $P \in \mathrm{GL}_n(\mathbb{Z})$ and some orthogonal matrix K.

Remarks.

(1) We have considered L in particular as a free discrete sub-$\mathbb{Z}$-module of the n-dimensional Euclidean space $L \otimes \mathbb{R}$. In many situations, it is desirable to consider L as a free discrete sub-$\mathbb{Z}$-module of some Euclidean space E of dimension m larger than n. The matrix B of coordinates of a basis of L on some orthonormal basis of E will then be an $m \times n$ matrix, but the Gram matrix $Q = B^tB$ will still be an $n \times n$ symmetric matrix.

(2) By abuse of language, we will frequently say that a free $\mathbb{Z}$-module of finite rank is a lattice even if there is no implicit quadratic form.

2.5.2 The Gram-Schmidt Orthogonalization Procedure

The existence of an orthonormal basis in a Euclidean vector space is often proved by using Gram-Schmidt orthonormalization (see any standard textbook). Doing this requires taking square roots, since the final vectors must be of length equal to 1.

For our purposes, we will need only an *orthogonal basis*, i.e. a set of mutually orthogonal vectors which are not necessarily of length 1. The same procedure works, except we do not normalize the length, and we will also call this the Gram-Schmidt orthogonalization procedure. It is summarized in the following proposition.

Proposition 2.5.4 (Gram-Schmidt). *Let $\mathbf{b}_i$ be a basis of a Euclidean vector space E. Define by induction:*

$$\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j}\mathbf{b}_j^* \qquad (1 \le i \le n),$$

where

$$\mu_{i,j} = \mathbf{b}_i \cdot \mathbf{b}_j^* / \mathbf{b}_j^* \cdot \mathbf{b}_j^* \qquad (1 \le j < i \le n),$$

then the $\mathbf{b}_i^*$ form an orthogonal (but not necessarily orthonormal) basis of E, $\mathbf{b}_i^*$ is the projection of $\mathbf{b}_i$ on the orthogonal complement of $\sum_{j=1}^{i-1} \mathbb{R}\mathbf{b}_j = \sum_{j=1}^{i-1} \mathbb{R}\mathbf{b}_j^*$, and the matrix M whose columns gives the coordinates of the $\mathbf{b}_i^*$ in terms of the $\mathbf{b}_i$ is an upper triangular matrix with diagonal terms equal to 1. In particular, if $d(L)$ is the determinant of the lattice L, we have $d(L)^2 = \prod_{1 \le i \le n} \|\mathbf{b}_i^*\|^2$.

The proof is trivial using induction. □

We will now give a number of corollaries of this construction.

Corollary 2.5.5 (Hadamard's Inequality). *Let (L, q) be a lattice of determinant $d(L)$, $(\mathbf{b}_i)_{1 \le i \le n}$ a $\mathbb{Z}$-basis of L, and for $x \in L$ write $|\mathbf{x}|$ for $q(\mathbf{x})^{1/2}$. Then*

$$d(L) \le \prod_{i=1}^{n} |\mathbf{b}_i|.$$

Equivalently, if B is an $n \times n$ matrix then

$$|\det(B)| \le \prod_{1 \le i \le n} \left(\sum_{1 \le j \le n} |b_{i,j}|^2 \right)^{1/2}.$$

Proof. If we set $B_i = |\mathbf{b}_i^*|^2$, the orthogonality of the $\mathbf{b}_i^*$ implies that

$$q(\mathbf{b}_i) = |\mathbf{b}_i|^2 = B_i + \sum_{1 \le j < i} \mu_{i,j}^2 B_j$$

hence $d(L)^2 = \prod_{1 \le i \le n} B_i \le \prod_{1 \le i \le n} |\mathbf{b}_i|^2$. □

Corollary 2.5.6. *Let B be an invertible matrix with coefficients in $\mathbb{R}$. Then there exist* unique *matrices K, A and N such that:*

(1) $B = KAN$.
(2) *K is an orthogonal matrix, in other words $K^t = K^{-1}$.*
(3) *A is a diagonal matrix with positive diagonal coefficients.*
(4) *N is an upper triangular matrix with diagonal terms equal to 1.*

Note that this Corollary is sometimes called the Iwasawa decomposition of B since it is in fact true in a much more general setting than that of the group $\mathrm{GL}_n(\mathbb{R})$.

Proof. Let B' be the matrix obtained by applying the Gram-Schmidt process to the vectors whose coordinates are the columns of B on the standard basis of $\mathbb{R}^n$. Then, by the proposition we have $B' = BN$ where N is an upper triangular matrix with diagonal terms equal to 1. Now the Gram-Schmidt process gives an orthogonal basis, in other words the Gram matrix of the $\mathbf{b}_i^*$ is a diagonal matrix D with positive entries. Let A be the diagonal matrix obtained from D by taking the positive square root of each coefficient (we will call A the square root of D). Then the equality ${B'}^t B' = D$ is equivalent to $B' = KA$ for an orthogonal matrix K, hence $BN = KA$ which is equivalent to the existence statement of the corollary.

The uniqueness statement also follows since the equality $B' = BN = KA$ means that the $\mathbf{b}_i'$ form an orthogonal basis which can be expressed on the $\mathbf{b}_i$ via an upper triangular matrix with diagonal terms equal to 1, and the procedure for obtaining this basis (i.e. the Gram-Schmidt coefficients) is clearly unique. □

Remarks.

(1) The requirement that the diagonal coefficients of A be positive is not essential, and is given only to insure uniqueness.
(2) By considering the inverse matrix and/or the transpose matrix of B, one has the same result with N lower triangular, or with $B = NAK$ instead of KAN.
(3) $T = AN$ is an upper triangular matrix with positive diagonal coefficients, and clearly any such upper triangular matrix T can be written uniquely in the form AN where A and N are as in the corollary. Hence we can use interchangeably both notations.

Another result is as follows.

Proposition 2.5.7. *If Q is the matrix of a positive definite quadratic form, then there exists a unique upper triangular matrix T with positive diagonal coefficients such that $Q = T^tT$ (or equivalently $Q = N^tDN$ where N is an upper triangular matrix with diagonal terms equal to 1 and D is a diagonal matrix with positive diagonal coefficients).*

Proof. By Proposition 2.5.3, we know that there exists $B \in \mathrm{GL}_n(\mathbb{R})$ such that $Q = B^tB$. On the other hand, by the Iwasawa decomposition we know that there exists matrices K and T such that $B = KT$ with K orthogonal and T upper triangular with positive diagonal coefficients ($T = AN$ in the notation of Proposition 2.5.6). Hence $Q = B^tB = T^tT$ thus showing the existence of T.

For the uniqueness, note that if $T^tT = T'^tT'$ with T and T' upper triangular, then

$$T'^{t^{-1}}T^t = T'T^{-1},$$

where taking inverses is justified since Q is a positive definite matrix. But the left hand side of this equality is a lower triangular matrix, while the right hand side is an upper triangular one, hence both sides must be equal to some diagonal matrix D, and plugging back in the initial equality and using again the invertibility of T, we obtain that D^2 is equal to the identity matrix. Now since the diagonal coefficients of $D = T'T^{-1}$ must be positive, we deduce that D itself is equal to the identity matrix, thus proving the proposition. □

We will give later an algorithm to find the matrix T (Algorithm 2.7.6).

2.6 Lattice Reduction Algorithms

2.6.1 The LLL Algorithm

Among all the $\mathbb{Z}$ bases of a lattice L, some are better than others. The ones whose elements are the shortest (for the corresponding norm associated to the quadratic form q) are called *reduced*. Since the bases all have the same determinant, to be reduced implies also that a basis is not too far from being orthogonal.

The notion of reduced basis is quite old, and in fact in some sense one can even define an optimal notion of reduced basis. The problem with this is that no really satisfactory algorithm is known to find such a basis in a reasonable time, except in dimension 2 (Algorithm 1.3.14), and quite recently in dimension 3 from the work of B. Vallée [Val].

A real breakthrough came in 1982 when A. K. Lenstra, H. W. Lenstra and L. Lovász succeeded in giving a new notion of reduction (what is now called

LLL-reduction) and simultaneously a reduction algorithm which is deterministic and polynomial time (see [LLL]). This has proved invaluable.

The LLL notion of reduction is as follows. Let $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$ be a basis of L. Using the Gram-Schmidt orthogonalization process, we can find an orthogonal (not orthonormal) basis $\mathbf{b}_1^*, \mathbf{b}_2^*, \ldots, \mathbf{b}_n^*$ as explained in Proposition 2.5.4.

Definition 2.6.1. *With the above notations, the basis* $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$ *is called* LLL-reduced *if*

$$|\mu_{i,j}| \leq \frac{1}{2} \qquad \text{for } 1 \leq j < i \leq n$$

and

$$|\mathbf{b}_i^* + \mu_{i,i-1}\mathbf{b}_{i-1}^*|^2 \geq \frac{3}{4}|\mathbf{b}_{i-1}^*|^2 \qquad \text{for } 1 < i \leq n,$$

or equivalently

$$|\mathbf{b}_i^*|^2 \geq \left(\frac{3}{4} - \mu_{i,i-1}^2\right)|\mathbf{b}_{i-1}^*|^2.$$

Note that the vectors $\mathbf{b}_i^* + \mu_{i,i-1}\mathbf{b}_{i-1}^*$ and $\mathbf{b}_{i-1}^*$ are the projections of $\mathbf{b}_i$ and $\mathbf{b}_{i-1}$ on the orthogonal complement of $\sum_{j=1}^{i-2} \mathbb{R}\mathbf{b}_j$.

Then we have the following theorem:

Theorem 2.6.2. *Let* $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$ *be an LLL-reduced basis of a lattice* L. *Then*

(1)
$$d(L) \leq \prod_{i=1}^{n} |\mathbf{b}_i| \leq 2^{n(n-1)/4} d(L),$$

(2)
$$|\mathbf{b}_j| \leq 2^{(i-1)/2}|\mathbf{b}_i^*|, \quad \text{if } 1 \leq j \leq i \leq n,$$

(3)
$$|\mathbf{b}_1| \leq 2^{(n-1)/4} d(L)^{1/n},$$

(4) *For every* $\mathbf{x} \in L$ *with* $\mathbf{x} \neq \mathbf{0}$ *we have*

$$|\mathbf{b}_1| \leq 2^{(n-1)/2}|\mathbf{x}|,$$

(5) *More generally, for any linearly independent vectors* $\mathbf{x}_1, \ldots, \mathbf{x}_t \in L$ *we have*

$$|\mathbf{b}_j| \leq 2^{(n-1)/2} \max(|\mathbf{x}_1|, \ldots, |\mathbf{x}_t|) \qquad \text{for } 1 \leq j \leq t.$$

We see that the vector $\mathbf{b}_1$ in a reduced basis is, in a very precise sense, not too far from being the shortest non-zero vector of L. In fact, it often is the shortest, and when it is not, one can, most of the time, work with $\mathbf{b}_1$ instead of the actual shortest vector.

Notation. In the rest of this chapter, we will use the notation $\mathbf{x} \cdot \mathbf{y}$ instead of $b(\mathbf{x}, \mathbf{y})$ where b is the bilinear form associated to q, and write $\mathbf{x}^2$ instead of $\mathbf{x} \cdot \mathbf{x} = q(\mathbf{x})$.

Proof. As in Corollary 2.5.5, we set $B_i = |\mathbf{b}_i^*|^2$. The first inequality of (1) is Corollary 2.5.5, Since the $\mathbf{b}_i$ are LLL-reduced, we have $B_i \geq (3/4 - \mu_{i,i-1}^2)B_{i-1} \geq B_{i-1}/2$ since $|\mu_{i,i-1}| \leq 1/2$. By induction, this shows that $B_j \leq 2^{i-j}B_i$ for $i \geq j$, hence

$$\mathbf{b}_i^2 \leq \frac{2^{i-1}+1}{2} B_i,$$

and this trivially implies Theorem 2.6.2 (1), in fact with a slightly better exponent of 2. Combining the two inequalities which we just obtained, we get for all $j \leq i$, $\mathbf{b}_j^2 \leq (2^{i-2} + 2^{i-j-1})B_i$ which implies (2). If we set $j = 1$ in (2) and take the product of (2) for $i = 1$ to $i = n$, we obtain $(\mathbf{b}_1^2)^n \leq 2^{n(n-1)/2} \prod_{1 \leq i \leq n} B_i = 2^{n(n-1)/2} d(L)^2$, proving (3). For (4), there exists an i such that $\mathbf{x} = \sum_{1 \leq j \leq i} r_j \mathbf{b}_j = \sum_{1 \leq j \leq i} s_j \mathbf{b}_j^*$ and $r_i \neq 0$, where $r_j \in \mathbb{Z}$ and $s_j \in \mathbb{R}$. It is clear from the definition of the $\mathbf{b}_j^*$ that $r_i = s_i$, hence

$$|\mathbf{x}|^2 \geq s_i^2 B_i = r_i^2 B_i \geq B_i$$

since r_i is a non-zero integer, and since by (2) we know that $B_i \geq 2^{1-i}|\mathbf{b}_1|^2 \geq 2^{1-n}|\mathbf{b}_1|^2$, (4) is proved. (5) is proved by a generalization of the present argument and is left to the reader. □

Remark. Although we have lost a little in the exponent of 2 in Theorem 2.6.2 (1), the proof shows that even using the optimal value given in our proof would not improve the estimate in (4). On the other hand, we have not completely used the full LLL-reduction inequalities. In particular, the inequalities on the $\mu_{i,j}$ can be weakened to $\mu_{i,j}^2 \leq 1/2$ for all $j < i - 1$ and $|\mu_{i,i-1}| \leq 1/2$. This can be used to speed up the reduction algorithm which follows.

As has already been mentioned, what makes all these notions and theorems so valuable is that there is a very simple and efficient algorithm to find a reduced basis in a lattice. We now describe this algorithm in its simplest form. The idea is as follows. Assume that the vectors $\mathbf{b}_1, \ldots, \mathbf{b}_{k-1}$ are already LLL-reduced (i.e. form an LLL-reduced basis of the lattice they generate). This will be initially the case for $k = 2$. The vector $\mathbf{b}_k$ first needs to be reduced so that $|\mu_{k,j}| \leq 1/2$ for all $j < k$ (some authors call this *size reduction*). This is done by replacing $\mathbf{b}_k$ by $\mathbf{b}_k - \sum_{j<k} a_j \mathbf{b}_j$ for some $a_j \in \mathbb{Z}$ in the following way. Assume that $|\mu_{k,j}| \leq 1/2$ for $l < j < k$ (initially with $l = k$). Then, if

$q = \lfloor \mu_{k,l} \rceil$ is the nearest integer to $\mu_{k,l}$, and, if we replace $\mathbf{b}_k$ by $\mathbf{b}_k - q\mathbf{b}_l$, then $\mu_{k,j}$ is not modified for $j > l$ (since $\mathbf{b}_j^*$ is orthogonal to $\mathbf{b}_l$ for $l < j$), and $\mu_{k,l}$ is replaced by $\mu_{k,l} - q$ (since $\mathbf{b}_l \cdot \mathbf{b}_l^* = \mathbf{b}_l^* \cdot \mathbf{b}_l^*$) and $|\mu_{k,l} - q| \leq 1/2$ hence the modified $\mu_{k,j}$ satisfy $|\mu_{k,j}| \leq 1/2$ for $l - 1 < j < k$.

Now that size reduction is done for the vector $\mathbf{b}_k$, we also need to satisfy the so-called *Lovász condition*, i.e. $B_k \geq (3/4 - \mu_{k,k-1}^2)B_{k-1}$. If this condition is satisfied, we increase k by 1 and start on the next vector $\mathbf{b}_k$ (if there is one). If it is not satisfied, we exchange the vectors $\mathbf{b}_k$ and $\mathbf{b}_{k-1}$, but then we must decrease k by 1 since we only know that $\mathbf{b}_1, \ldots, \mathbf{b}_{k-2}$ is LLL-reduced. A priori it is not clear that this succession of increments and decrements of k will ever terminate, but we will prove that this is indeed the case (and that the number of steps is not large) after giving the algorithm.

We could compute all the Gram-Schmidt coefficients $\mu_{k,j}$ and B_k at the beginning of the algorithm, and then update them during the algorithm. After each exchange step however, the coefficients $\mu_{i,k}$ and $\mu_{i,k-1}$ for $i > k$ must be updated, and this is usually a waste of time since they will probably change before they are used. Hence, it is a better idea to compute the Gram-Schmidt coefficients as needed, keeping in a variable $k_{\max}$ the maximal value of k that has been attained.

Another improvement on the basic idea is to reduce only the coefficient $\mu_{k,k-1}$ and not all the $\mu_{k,l}$ for $l < k$ during size-reduction, since this is the only coefficient which must be less than $1/2$ in absolute value before testing the Lovász condition. All this leads to the following algorithm.

Algorithm 2.6.3 (LLL Algorithm). Given a basis $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$ of a lattice (L, q) (either by coordinates on the canonical basis of $\mathbb{R}^m$ for some $m \geq n$ or by its Gram matrix), this algorithm transforms the vectors $\mathbf{b}_i$ so that when the algorithm terminates, the $\mathbf{b}_i$ form an LLL-reduced basis. In addition, the algorithm outputs a matrix H giving the coordinates of the LLL-reduced basis in terms of the initial basis. As usual we will denote by H_i the columns of H.

1. [Initialize] Set $k \leftarrow 2$, $k_{\max} \leftarrow 1$, $\mathbf{b}_1^* \leftarrow \mathbf{b}_1$, $B_1 \leftarrow \mathbf{b}_1 \cdot \mathbf{b}_1$ and $H \leftarrow I_n$.
2. [Incremental Gram-Schmidt] If $k \leq k_{\max}$ go to step 3. Otherwise, set $k_{\max} \leftarrow k$, $\mathbf{b}_k^* \leftarrow \mathbf{b}_k$, then for $j = 1, \ldots, k-1$ set $\mu_{k,j} \leftarrow \mathbf{b}_k \cdot \mathbf{b}_j^* / B_j$ and $\mathbf{b}_k^* \leftarrow \mathbf{b}_k^* - \mu_{k,j}\mathbf{b}_j^*$. Finally, set $B_k \leftarrow \mathbf{b}_k^* \cdot \mathbf{b}_k^*$ (see Remark (2) below for the corresponding step if only the Gram matrix of the $\mathbf{b}_i$ is given). If $B_k = 0$ output an error message saying that the $\mathbf{b}_i$ did not form a basis and terminate the algorithm.
3. [Test LLL condition] Execute Sub-algorithm RED$(k, k-1)$ below. If $B_k < (0.75 - \mu_{k,k-1}^2)B_{k-1}$, execute Sub-algorithm SWAP(k) below, set $k \leftarrow \max(2, k-1)$ and go to step 3. Otherwise, for $l = k-2, k-3, \ldots, 1$ execute Sub-algorithm RED(k, l), then set $k \leftarrow k+1$.
4. [Finished?] If $k \leq n$, then go to step 2. Otherwise, output the LLL reduced basis $\mathbf{b}_i$, the transformation matrix $H \in \mathrm{GL}_n(\mathbb{Z})$ and terminate the algorithm.

Sub-algorithm RED(k, l). If $|\mu_{k,l}| \leq 0.5$ terminate the sub-algorithm. Otherwise, let q be the integer nearest to $\mu_{k,l}$, i.e.

$$q \leftarrow \lfloor \mu_{k,l} \rceil = \lfloor 0.5 + \mu_{k,l} \rfloor.$$

Set $\mathbf{b}_k \leftarrow \mathbf{b}_k - q\mathbf{b}_l$, $H_k \leftarrow H_k - qH_l$, $\mu_{k,l} \leftarrow \mu_{k,l} - q$, for all i such that $1 \le i \le l-1$, set $\mu_{k,i} \leftarrow \mu_{k,i} - q\mu_{l,i}$ and terminate the sub-algorithm.

Sub-algorithm SWAP(k). Exchange the vectors $\mathbf{b}_k$ and $\mathbf{b}_{k-1}$, H_k and H_{k-1}, and if $k > 2$, for all j such that $1 \le j \le k-2$ exchange $\mu_{k,j}$ with $\mu_{k-1,j}$. Then set (in this order) $\mu \leftarrow \mu_{k,k-1}$, $B \leftarrow B_k + \mu^2 B_{k-1}$, $\mu_{k,k-1} \leftarrow \mu B_{k-1}/B$, $\mathbf{b} \leftarrow \mathbf{b}^*_{k-1}$, $\mathbf{b}^*_{k-1} \leftarrow \mathbf{b}^*_k + \mu\mathbf{b}$, $\mathbf{b}^*_k \leftarrow -\mu_{k,k-1}\mathbf{b}^*_k + (B_k/B)\mathbf{b}$, $B_k \leftarrow B_{k-1}B_k/B$ and $B_{k-1} \leftarrow B$. Finally, for $i = k+1, k+2, \ldots, k_{\max}$ set (in this order) $t \leftarrow \mu_{i,k}$, $\mu_{i,k} \leftarrow \mu_{i,k-1} - \mu t$, $\mu_{i,k-1} \leftarrow t + \mu_{k,k-1}\mu_{i,k}$ and terminate the sub-algorithm.

Proof. It is easy to show that at the beginning of step 4, the LLL conditions of Definition 2.6.1 are valid for $i \le k-1$. Hence, if $k > n$, we have indeed obtained an LLL-reduced family, and since it is clear that the operations which are performed on the $\mathbf{b}_i$ are of determinant ± 1, this family is a basis of L, hence the output of the algorithm is correct. What we must show is that the algorithm does in fact terminate.

If we set for $0 \le i \le n$

$$d_i = \det\left((\mathbf{b}_r \cdot \mathbf{b}_s)_{1 \le r,s \le i}\right),$$

we easily check that

$$d_i = \prod_{1 \le j \le i} B_j,$$

where as usual $B_i = |\mathbf{b}^*_i|^2$, and in particular $d_i > 0$, and it is clear from this that $d_0 = 1$ and $d_n = d(L)^2$. Set

$$D = \prod_{1 \le i \le n-1} d_i.$$

This can change only if some B_i changes, and this can occur only in Sub-algorithm SWAP. In that sub-algorithm the d_i are unchanged for $i < k-1$ and for $i \ge k$, and by the condition of step 3, d_{k-1} is multiplied by a factor at most equal to $3/4$. Hence D is also reduced by a factor at most equal to $3/4$. Let L_i be the lattice of dimension i generated by the $\mathbf{b}_j$ for $j \le i$, and let s_i be the smallest non-zero value of the quadratic form q in L_i. Using Proposition 6.4.1 which we will give in Chapter 6, we obtain

$$d_i \ge s_i^i \gamma_i^{-i} \ge s_n^i \gamma_i^{-i},$$

and since s_n is the smallest non-zero value of $q(x)$ on L, this last expression depends only on i but not on the $\mathbf{b}_j$. It follows that d_i is bounded from below by a positive constant depending only on i and L. Hence D is bounded from below by a positive constant depending only on L, and this shows that the number of times that Sub-algorithm SWAP is executed must be finite.

Since this is the only place where k can decrease (after execution of the sub-algorithm) the algorithm must terminate, and this finishes the proof of its validity. □

A more careful analysis shows that the running time of the LLL algorithm is at most $O(n^6 \ln^3 B)$, if $|\mathbf{b}_i|^2 \leq B$ for all i. In practice however, this upper bound is quite pessimistic.

Remarks.

(1) If the matrix transformation H is not desired, one can suppress from the algorithm all the statements concerning it, since it does not play any real role.

(2) On the other hand if the $\mathbf{b}_i$ are given only by their Gram matrix, the $\mathbf{b}_i$ and $\mathbf{b}_i^*$ exist only abstractly. Hence, the only output of the algorithm is the matrix H, and the updating of the vectors $\mathbf{b}_i$ done in Sub-algorithms RED and SWAP must be done directly on the Gram matrix.

In particular, step 2 must then be replaced as follows (see Exercise 21).

2. [Incremental Gram-Schmidt] If $k \leq k_{\max}$ go to step 3. Otherwise, set $k_{\max} \leftarrow k$ then for $j = 1, \ldots, k-1$ set $a_{k,j} \leftarrow \mathbf{b}_k \cdot \mathbf{b}_j - \sum_{i=1}^{j-1} \mu_{j,i} a_{k,i}$ and $\mu_{k,j} \leftarrow a_{k,j}/B_j$, then set $B_k \leftarrow \mathbf{b}_k \cdot \mathbf{b}_k - \sum_{i=1}^{k-1} \mu_{k,i} a_{k,i}$. If $B_k = 0$ output an error message saying that the $\mathbf{b}_i$ did not form a basis and terminate the algorithm.

The auxiliary array $a_{k,j}$ is used to minimize the number of operations, otherwise we could of course write the formulas directly with $\mu_{k,j}$.

Asymptotically, this requires $n^3/6$ multiplications/divisions, and this is much faster than the $n^2 m/2$ required by Gram-Schmidt when only the coordinates of the $\mathbf{b}_i$ are known. Since the computation of the Gram matrix from the coordinates of the $\mathbf{b}_i$ also requires asymptotically $n^2 m/2$ multiplications, one should use directly the formulas of Algorithm 2.6.3 when the Gram matrix is not given.

(3) The constant 0.75 in step 3 of the algorithm can be replaced by any constant c such that $1/4 < c < 1$. Of course, this changes the estimates given by Theorem 2.6.2. (In the results and proof of the theorem, replace 2 by $\alpha = 1/(c - 1/4)$, and use the weaker inequality $\mu_{k,l}^2 \leq (\alpha - 1)/\alpha$).) The speed of the algorithm and the "quality" of the final basis which one obtains, are relatively insensitive to the value of the constant. In practice, one should perhaps use $c = 0.99$. The ideal value would be $c = 1$, but in this case one does not know whether the LLL algorithm runs in polynomial time, although in practice this seems to be the case.

(4) Another possibility, suggested by LaMacchia in [LaM] is to *vary* the constant c in the course of the algorithm, starting the reduction with a constant c slightly larger than $1/4$ (so that the reduction is as fast as possible), and increasing it so as to reach $c = 0.99$ at the end of the reduction, so

that the quality of the reduced basis is a good as possible. We refer to [LaM] for details.

(5) We can also replace the LLL condition $B_k \geq (3/4 - \mu_{k,k-1}^2)B_{k-1}$ by the so-called Siegel condition $B_k \geq B_{k-1}/2$. Indeed, since $|\mu_{k,k-1}| \leq 1/2$, the LLL condition with the constant $c = 3/4$ implies the Siegel condition, and conversely the Siegel condition implies the LLL condition for the constant $c = 1/2$. In that case the preliminary reduction RED$(k, k-1)$ should be performed after the test, together with the other RED(k, l).

(6) If the Gram matrix does not necessarily have rational coefficients, the $\mu_{i,j}$ and B_i must be represented approximately using floating point arithmetic. Even if the Gram matrix is rational or even integral, it is often worthwhile to work using floating point arithmetic. The main problem with this approach is that roundoff errors may prevent the final basis from being LLL reduced. In many cases, this is not really important since the basis is not far from being LLL reduced. It may happen however that the roundoff errors cause catastrophic divergence from the LLL algorithm, and consequently give a basis which is very far from being reduced in any sense. Hence we must be careful. Let r be the number of relative precision bits.

First, during step 2 it is possible to replace the computation of the products $\mathbf{b}_i \cdot \mathbf{b}_j$ by floating point approximations (of course only in the case where the $\mathbf{b}_i$ are given by coordinates, otherwise there is nothing to compute). This should not be done if $\mathbf{b}_i$ and $\mathbf{b}_j$ are nearly orthogonal, i.e. if $\mathbf{b}_i \cdot \mathbf{b}_j/|\mathbf{b}_i||\mathbf{b}_j|$ is smaller than $2^{-r/2}$ say. In that case, $\mathbf{b}_i \cdot \mathbf{b}_j$ should be computed as exactly as possible using the given data.

Second, at the beginning of Sub-algorithm RED, the nearest integer q to $\mu_{k,l}$ is computed. If q is too large, say $q > 2^{r/2}$, then $\mu_{k,l} - q$ will have a small relative precision and the values of the $\mu_{k,l}$ will soon become incorrect. In that case, we should recompute the $\mu_{k,l}$, $\mu_{k-1,l}$, B_{k-1} and B_k directly from the Gram-Schmidt formulas, set $k \leftarrow \max(k-1, 2)$ and start again at step 3.

These modifications (and many more) are explained in a rigorous theoretical setting in [Schn], and for practical uses in [Schn-Euch] to which we refer.

(7) The algorithm assumes that the $\mathbf{b}_i$ are linearly independent. If they are not, we will get an error message in the Gram-Schmidt stage of the algorithm. It is possible to modify the algorithm so that it will not only work in this case, but in fact output a true basis and a set of linearly independent relations for the initial set of vectors (see Algorithm 2.6.8).

2.6.2 The LLL Algorithm with Deep Insertions

A modification of the LLL algorithm due to Schnorr and Euchner ([Schn-Euc]) is the following. It may be argued that the Lovász condition $B_k \geq (0.75 - \mu_{k,k-1}^2)B_{k-1}$ (in addition to the requirement $\mu_{k,j} \leq 1/2$) should be

strengthened, taking into account the earlier B_j. If this is done rashly however, it leads to a non-polynomial time algorithm, both in theory and in practice. This is, of course, one of the reasons for the choice of a weaker condition. Schnorr and Euchner (loc. cit.) have observed however that one can strengthen the above condition without losing much on the practical speed of the algorithm, although in the worst case the resulting algorithm is no longer polynomial time. They report that in many cases, this leads to considerably shorter lattice vectors than the basic LLL algorithm.

The idea is as follows. If $\mathbf{b}_k$ is inserted between $\mathbf{b}_{i-1}$ and $\mathbf{b}_i$ for some $i < k$, then (Exercise 22) the new B_i will become

$$\mathbf{b}_k \cdot \mathbf{b}_k - \sum_{1 \le j < i} \mu_{k,j}^2 B_j = B_k + \sum_{i \le j < k} \mu_{k,j}^2 B_j .$$

If this is significantly smaller than the old B_i (say at most $\frac{3}{4}B_i$ as in our initial version of LLL), then it is reasonable to do this insertion. Note that the case $i = k - 1$ of this test is exactly the original LLL condition. For these tests to make sense, Algorithm RED(k, l) must be executed before the test for all $l < k$ and not only for $l = k - 1$ as in Algorithm 2.6.3.

Inserting $\mathbf{b}_k$ just after $\mathbf{b}_{i-1}$ for some $i < k$ will be called a *deep insertion*. After such an insertion, k must be set back to $\max(i - 1, 2)$, and the $\mu_{j,l}$ and B_j must be updated. When $i < k - 1$ however, the formulas become complicated and it is probably best to recompute the new Gram-Schmidt coefficients instead of updating them. One consequence of this is that we do not need to keep track of the largest value $k_{\max}$ that k has attained.

This leads to the following algorithm, due in essence to Schnorr and Euchner ([Schn-Euc]).

Algorithm 2.6.4 (LLL Algorithm with Deep Insertions). Given a basis $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$ of a lattice (L, q) (either by coordinates in the canonical basis of $\mathbb{R}^m$ for some $m \ge n$ or by its Gram matrix), this algorithm transforms the vectors $\mathbf{b}_i$ so that when the algorithm terminates, the $\mathbf{b}_i$ form an LLL-reduced basis. In addition, the algorithm outputs a matrix H giving the coordinates of the LLL-reduced basis in terms of the initial basis. As usual we will denote by H_i the columns H.

1. [Initialize] Set $k \leftarrow 1$ and $H \leftarrow I_n$.
2. [Incremental Gram-Schmidt] Set $\mathbf{b}_k^* \leftarrow \mathbf{b}_k$, then for $j = 1, \ldots, k-1$ set $\mu_{k,j} \leftarrow \mathbf{b}_k \cdot \mathbf{b}_j^* / B_j$ and $\mathbf{b}_k^* \leftarrow \mathbf{b}_k^* - \mu_{k,j}\mathbf{b}_j^*$. Then set $B_k \leftarrow \mathbf{b}_k^* \cdot \mathbf{b}_k^*$. If $B_k = 0$ output an error message saying that the $\mathbf{b}_i$ did not form a basis and terminate the algorithm. Finally, if $k = 1$, set $k \leftarrow 2$ and go to step 5.
3. [Initialize test] For $l = k - 1, k - 2, \ldots, 1$ execute Sub-algorithm RED(k, l) above. Set $B \leftarrow \mathbf{b}_k \cdot \mathbf{b}_k$ and $i \leftarrow 1$.
4. [Deep LLL test] If $i = k$, set $k \leftarrow k + 1$ and go to step 5. Otherwise, do as follows. If $\frac{3}{4}B_i \le B$ set $B \leftarrow B - \mu_{k,i}^2 B_i$, $i \leftarrow i + 1$ and go to step 4.

Otherwise, execute Algorithm INSERT(k,i) below. If $i \geq 2$ set $k \leftarrow i-1$, $B \leftarrow \mathbf{b}_k \cdot \mathbf{b}_k$, $i \leftarrow 1$ and go to step 4. If $i = 1$, set $k \leftarrow 1$ and go to step 2.

5. [Finished?] If $k \leq n$, then go to step 2. Otherwise, output the LLL reduced basis $\mathbf{b}_i$, the transformation matrix $H \in \mathrm{GL}_n(\mathbb{Z})$ and terminate the algorithm.

Sub-algorithm INSERT(k,i). Set $\mathbf{b} \leftarrow \mathbf{b}_k$, $V \leftarrow H_k$, for $j = k, k-1, \ldots, i+1$ set $\mathbf{b}_j \leftarrow \mathbf{b}_{j-1}$ and $H_j \leftarrow H_{j-1}$, and finally set $\mathbf{b}_i \leftarrow \mathbf{b}$ and $H_i \leftarrow V$. Terminate the sub-algorithm.

2.6.3 The Integral LLL Algorithm

If the Gram matrix of the $\mathbf{b}_i$ has integral coefficients, the $\mu_{i,j}$ and the B_k will be rational and it may be tempting to do all the computation with rational numbers. Unfortunately, the repeated GCD computations necessary for performing rational arithmetic during the algorithm slows it down considerably. There are essentially two ways to overcome this problem. The first is to do only approximate computations of the $\mu_{i,j}$ and the B_i as mentioned above.

The second is as follows. In the proof of Algorithm 2.6.3 we have introduced quantities d_i which are clearly integral in our case, since they are equal to subdeterminants of our Gram matrix. We have the following integrality results.

Proposition 2.6.5. *Assume that the Gram matrix* $(\mathbf{b}_i \cdot \mathbf{b}_j)$ *is integral, and set*

$$d_i = \det((\mathbf{b}_r \cdot \mathbf{b}_s)_{1 \leq r,s \leq i}) = \prod_{1 \leq j \leq i} B_j.$$

Then for all i *and for all* $j < i$

(1)

$$d_{i-1}B_i \in \mathbb{Z} \qquad \text{and} \qquad d_j\mu_{i,j} \in \mathbb{Z}.$$

(2) *for all* m *such that* $j < m \leq i$

$$d_j \sum_{1 \leq k \leq j} \mu_{i,k}\mu_{m,k}B_k \in \mathbb{Z}.$$

Proof. We have seen above that $d_i = \prod_{1 \leq k \leq i} B_k$ hence $d_{i-1}B_i = d_i \in \mathbb{Z}$. For the second statement of (1), let $j < i$ and consider the vector

$$\mathbf{v} = \mathbf{b}_i - \sum_{1 \leq k \leq j} \mu_{i,k}\mathbf{b}_k^* = \mathbf{b}_i^* + \sum_{j < k < i} \mu_{i.k}\mathbf{b}_k^*.$$

From the second expression it is clear that $\mathbf{b}_k^* \cdot \mathbf{v} = 0$ for all k such that $1 \leq k \leq j$, or equivalently since the $\mathbb{R}$-span of the $\mathbf{b}_k^*$ $(1 \leq k \leq j)$ is equal to the $\mathbb{R}$-span of the $\mathbf{b}_k$,

$$\mathbf{b}_k \cdot \mathbf{v} = 0 \quad \text{for} \quad 1 \le k \le j.$$

For the same reason, we can write

$$\mathbf{v} = \mathbf{b}_i - \sum_{1 \le k \le j} x_k \mathbf{b}_k$$

for some $x_k \in \mathbb{R}$. Then the above equations can be written in matrix form

$$\begin{pmatrix} \mathbf{b}_1 \cdot \mathbf{b}_1 & \dots & \mathbf{b}_1 \cdot \mathbf{b}_j \\ \vdots & \ddots & \vdots \\ \mathbf{b}_j \cdot \mathbf{b}_1 & \dots & \mathbf{b}_j \cdot \mathbf{b}_j \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_j \end{pmatrix} = \begin{pmatrix} \mathbf{b}_i \cdot \mathbf{b}_1 \\ \vdots \\ \mathbf{b}_i \cdot \mathbf{b}_j \end{pmatrix}.$$

In particular, since the determinant of the matrix is equal by definition to d_j, by inverting the matrix we see that the x_k are of the form m_k/d_j for some $m_k \in \mathbb{Z}$ (since the Gram matrix is integral). Furthermore, the equality $\sum_{1 \le k \le j} x_k \mathbf{b}_k = \sum_{1 \le k \le j} \mu_{i,k} \mathbf{b}_k^*$ shows by projection on $\mathbf{b}_j^*$ that $x_j = \mu_{i,j}$, thus proving (1).

For (2) we note that by what we have proved, $d_j \mathbf{v}$ is an integral linear combination of the $\mathbf{b}_k$ (in other words it belongs to the lattice), hence in particular $d_j \mathbf{v} \cdot \mathbf{b}_m \in \mathbb{Z}$ for all m such that $1 \le m \le n$. Since $\mathbf{v} = \mathbf{b}_i - \sum_{1 \le k \le j} \mu_{i,k} \mathbf{b}_k^*$, we obtain (2). □

Corollary 2.6.6. *With the same hypotheses and notations as the proposition, set $\lambda_{i,j} = d_j \mu_{i,j}$ for $j < i$ (so $\lambda_{i,j} \in \mathbb{Z}$) and $\lambda_{i,i} = d_i$. Then for $j \le i$ fixed, if we define the sequence u_k by $u_0 = \mathbf{b}_i \cdot \mathbf{b}_j$ and for $1 \le k < j$*

$$u_k = \frac{d_k u_{k-1} - \lambda_{i,k} \lambda_{j,k}}{d_{k-1}},$$

then $u_k \in \mathbb{Z}$ and $u_{j-1} = \lambda_{i,j}$.

Proof. It is easy to check by induction on k that

$$u_k = d_k \left(\mathbf{b}_i \cdot \mathbf{b}_j - \sum_{1 \le l \le k} \frac{\lambda_{i,l} \lambda_{j,l}}{d_l d_{l-1}} \right) = d_k \left(\mathbf{b}_i \cdot \mathbf{b}_j - \sum_{1 \le l \le k} \mu_{i,l} \mu_{j,l} B_l \right)$$

and the proposition shows that this last expression is integral. We also have $u_{j-1} = B_j d_{j-1} \mu_{i,j} = d_j \mu_{i,j} = \lambda_{i,j}$ thus proving the corollary. □

Using these results, it is easy to modify Algorithm 2.6.3 so as to work entirely with integers. This leads to the following algorithm, where it is assumed that the basis is given by its Gram-Schmidt matrix. (Hence, if the basis is given in terms of coordinates, compute first the Gram-Schmidt matrix before applying the algorithm, or modify appropriately the formulas of step 1.) Essentially the same algorithm is given in [de Weg].

Algorithm 2.6.7 (Integral LLL Algorithm). Given a basis $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$ of a lattice (L, q) by its Gram matrix which is assumed to have integral coefficients, this algorithm transforms the vectors $\mathbf{b}_i$ so that when the algorithm terminates, the $\mathbf{b}_i$ form an LLL-reduced basis. The algorithm outputs a matrix H giving the coordinates of the LLL-reduced basis in terms of the initial basis. We will denote by H_i the column vectors of H. All computations are done using integers only.

1. [Initialize] Set $k \leftarrow 2$, $k_{\max} \leftarrow 1$, $d_0 \leftarrow 1$, $d_1 \leftarrow \mathbf{b}_1 \cdot \mathbf{b}_1$ and $H \leftarrow I_n$.

2. [Incremental Gram-Schmidt] If $k \leq k_{\max}$ go to step 3. Otherwise, set $k_{\max} \leftarrow k$ and for $j = 1, \ldots, k$ (in that order) do as follows: set $u \leftarrow \mathbf{b}_k \cdot \mathbf{b}_j$ and for $i = 1, \ldots, j-1$ set
$$u \leftarrow \frac{d_i u - \lambda_{k,i}\lambda_{j,i}}{d_{i-1}}$$
(the result is in $\mathbb{Z}$), then if $j < k$ set $\lambda_{k,j} \leftarrow u$ and if $j = k$ set $d_k \leftarrow u$. If $d_k = 0$, the $\mathbf{b}_i$ did not form a basis, hence output an error message and terminate the algorithm (but see also Algorithm 2.6.8).

3. [Test LLL condition] Execute Sub-algorithm REDI$(k, k-1)$ below. If $d_k d_{k-2} < \frac{3}{4}d_{k-1}^2 - \lambda_{k,k-1}^2$, execute algorithm SWAPI(k) below, set $k \leftarrow \max(2, k-1)$ and go to step 3. Otherwise, for $l = k-2, k-3, \ldots, 1$ execute Sub-algorithm REDI(k, l), then set $k \leftarrow k+1$.

4. [Finished?] If $k \leq n$ go to step 2. Otherwise, output the transformation matrix $H \in \mathrm{GL}_n(\mathbb{Z})$ and terminate the algorithm.

Sub-algorithm REDI(k, l). If $|2\lambda_{k,l}| \leq d_l$ terminate the sub-algorithm. Otherwise, let q be the integer nearest to $\lambda_{k,l}/d_l$, i.e. the quotient of the Euclidean division of $2\lambda_{k,l}+d_l$ by $2d_l$. Set $H_k \leftarrow H_k - qH_l$, $\mathbf{b}_k \leftarrow \mathbf{b}_k - q\mathbf{b}_l$, $\lambda_{k,l} \leftarrow \lambda_{k,l} - qd_l$, for all i such that $1 \leq i \leq l-1$ set $\lambda_{k,i} \leftarrow \lambda_{k,i} - q\lambda_{l,i}$ and terminate the sub-algorithm.

Sub-algorithm SWAPI(k). Exchange the vectors H_k and H_{k-1}, exchange $\mathbf{b}_k$ and $\mathbf{b}_{k-1}$, and if $k > 2$, for all j such that $1 \leq j \leq k-2$ exchange $\lambda_{k,j}$ with $\lambda_{k-1,j}$. Then set $\lambda \leftarrow \lambda_{k,k-1}$, $B \leftarrow (d_{k-2}d_k + \lambda^2)/d_{k-1}$, then for $i = k+1, k+2, \ldots k_{\max}$ set (in this order) $t \leftarrow \lambda_{i,k}$, $\lambda_{i,k} \leftarrow (d_k\lambda_{i,k-1} - \lambda t)/d_{k-1}$ and $\lambda_{i,k-1} \leftarrow (Bt + \lambda\lambda_{i,k})/d_k$. Finally, set $d_{k-1} \leftarrow B$ and terminate the sub-algorithm.

It is an easy exercise (Exercise 24) to check that these formulas correspond exactly to the formulas of Algorithm 2.6.3.

Remark. In step 3, the fundamental LLL comparison $d_k d_{k-2} < \frac{3}{4}d_{k-1}^2 - \lambda_{k,k-1}^2$ involves the non-integral number $\frac{3}{4}$ (it could also be 0.99). This is not really a problem since this comparison can be done any way one likes (by multiplying by 4, or using floating point arithmetic), since a roundoff error at that point is totally unimportant.

2.6.4 LLL Algorithms for Linearly Dependent Vectors

As has been said above, the LLL algorithm cannot be applied directly to a system of linearly dependent vectors $\mathbf{b}_i$. It can however be modified so as to work in this case, and to output a basis and a system of relations. The problem is that in the Gram-Schmidt orthogonalization procedure we will have at some point $B_i = \mathbf{b}_i^* \cdot \mathbf{b}_i^* = 0$. This means of course that $\mathbf{b}_i$ is equal to a linear combination of the $\mathbf{b}_j$ for $j < i$. Since Gram-Schmidt performs projections of the successive vectors on the subspace generated by the preceding ones, this means that we can forget the index i in the rest of the orthogonalization (although not the vector $\mathbf{b}_i$ itself). This leads to the following algorithm which is very close to Algorithm 2.6.3 and whose proof is left to the reader.

Algorithm 2.6.8 (LLL Algorithm on Not Necessarily Independent Vectors). Given n non-zero vectors $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$ generating a lattice (L, q) (either by coordinates or by their Gram matrix), this algorithm transforms the vectors $\mathbf{b}_i$ and computes the rank p of the lattice L so that when the algorithm terminates $\mathbf{b}_i = 0$ for $1 \leq i \leq n-p$ and the $\mathbf{b}_i$ for $n-p < i \leq n$ form an LLL-reduced basis of L. In addition, the algorithm outputs a matrix H giving the coordinates of the new $\mathbf{b}_i$ in terms of the initial ones. In particular, the first $n-p$ columns H_i of H will be a basis of relation vectors for the $\mathbf{b}_i$, i.e. of vectors $\mathbf{r}$ such that $\sum_{1 \leq i \leq n} r_i \mathbf{b}_i = 0$.

1. [Initialize] Set $k \leftarrow 2$, $k_{\max} \leftarrow 1$, $\mathbf{b}_1^* \leftarrow \mathbf{b}_1$, $B_1 \leftarrow \mathbf{b}_1 \cdot \mathbf{b}_1$ and $H \leftarrow I_n$.

2. [Incremental Gram-Schmidt] If $k \leq k_{\max}$ go to step 3. Otherwise, set $k_{\max} \leftarrow k$ and for $j = 1, \ldots, k-1$ set $\mu_{k,j} \leftarrow \mathbf{b}_k \cdot \mathbf{b}_j^* / B_j$ if $B_j \neq 0$ and $\mu_{k,j} \leftarrow 0$ if $B_j = 0$, then set $\mathbf{b}_k^* \leftarrow \mathbf{b}_k - \sum_{j=1}^{k-1} \mu_{k,j} \mathbf{b}_j^*$ and $B_k \leftarrow \mathbf{b}_k^* \cdot \mathbf{b}_k^*$ (use the formulas given in Remark (2) above if the $\mathbf{b}_i$ are given by their Gram matrix).

3. [Test LLL condition] Execute Sub-algorithm RED$(k, k-1)$ above. If $B_k < (0.75 - \mu_{k,k-1}^2) B_{k-1}$, execute Sub-algorithm SWAPG(k) below, set $k \leftarrow \max(2, k-1)$ and go to step 3. Otherwise, for $l = k-2, k-3, \ldots, 1$ execute Sub-algorithm RED(k, l), then set $k \leftarrow k+1$.

4. [Finished?] If $k \leq n$ go to step 2. Otherwise, let r be the number of initial vectors $\mathbf{b}_i$ which are equal to zero, output $p \leftarrow n - r$, the vectors $\mathbf{b}_i$ for $r+1 \leq i \leq n$ (which form an LLL-reduced basis of L), the transformation matrix $H \in \mathrm{GL}_n(\mathbb{Z})$ and terminate the algorithm.

Sub-algorithm SWAPG(k). Exchange the vectors $\mathbf{b}_k$ and $\mathbf{b}_{k-1}$, H_k and H_{k-1}, and if $k > 2$, for all j such that $1 \leq j \leq k-2$ exchange $\mu_{k,j}$ with $\mu_{k-1,j}$. Then set $\mu \leftarrow \mu_{k,k-1}$ and $B \leftarrow B_k + \mu^2 B_{k-1}$. Now, in the case $B = 0$ (i.e. $B_k = \mu = 0$), exchange B_k and B_{k-1}, exchange $\mathbf{b}_k^*$ and $\mathbf{b}_{k-1}^*$ and for $i = k+1, k+2, \ldots k_{\max}$ exchange $\mu_{i,k}$ and $\mu_{i,k-1}$.

In the case $B_k = 0$ and $\mu \neq 0$, set $B_{k-1} \leftarrow B$, $\mathbf{b}_{k-1}^* \leftarrow \mu \mathbf{b}_{k-1}^*$, $\mu_{k,k-1} \leftarrow 1/\mu$ and for $i = k+1, k+2, \ldots, k_{\max}$ set $\mu_{i,k-1} \leftarrow \mu_{i,k-1}/\mu$.

Finally, in the case $B_k \neq 0$, set (in this order) $t \leftarrow B_{k-1}/B$, $\mu_{k,k-1} \leftarrow \mu t$,

$\mathbf{b} \leftarrow \mathbf{b}_{k-1}^*$, $\mathbf{b}_{k-1}^* \leftarrow \mathbf{b}_k^* + \mu\mathbf{b}$, $\mathbf{b}_k^* \leftarrow -\mu_{k,k-1}\mathbf{b}_k^* + (B_k/B)\mathbf{b}$, $B_k \leftarrow B_k t$, $B_{k-1} \leftarrow B$, then for $i = k+1, k+2, \ldots, k_{\max}$ set (in this order) $t \leftarrow \mu_{i,k}$, $\mu_{i,k} \leftarrow \mu_{i,k-1} - \mu t$, $\mu_{i,k-1} \leftarrow t + \mu_{k,k-1}\mu_{i,k}$. Terminate the sub-algorithm.

Note that in this sub-algorithm, in the case $B = 0$, we have $B_k = 0$ and hence $\mu_{i,k} = 0$ for $i > k$, so the exchanges are equivalent to setting $B_k \leftarrow B_{k-1}$, $B_{k-1} \leftarrow 0$ and for $i \geq k+1$, $\mu_{i,k} \leftarrow \mu_{i,k-1}$ and $\mu_{i,k-1} \leftarrow 0$.

An important point must be made concerning this algorithm. Since several steps of the algorithm test whether some quantity is equal to zero or not, it can be applied only to vectors with exact (i.e. rational) entries. Indeed, for vectors with non-exact entries, the notion of relation vector is itself not completely precise since some degree of approximation must be given in advance. Thus the reader is advised to use caution when using LLL algorithms for linearly dependent vectors when they are non-exact. (For instance, we could replace a test $B_k = 0$ by $B_k \leq \varepsilon$ for a suitable ε.)

We must prove that this algorithm is valid. To show that it terminates, we use a similar quantity to the one used in the proof of the validity of Algorithm 2.6.3. We set

$$d_k = \prod_{i \leq k, B_i \neq 0} B_i \quad \text{and} \quad D = \prod_{k \leq n, B_k \neq 0} d_k \prod_{k \leq n, B_k = 0} 2^k.$$

This quantity is modified only in Sub-algorithm SWAPG(k). If $B = B_k + \mu^2 B_{k-1} \neq 0$, then d_{k-1} is multiplied by a factor which is smaller than $3/4$ and the others are unchanged, hence D decreases by a factor at least $3/4$ as in the usual LLL algorithm. If $B = 0$, then B_{k-1} becomes 0 and B_k becomes equal to B_{k-1}, hence d_{k-1} becomes equal to d_{k-2}, d_k stays the same (since $B_{k-1}d_{k-2} = d_{k-1} = d_k$ when $B_k = 0$) as well as the others, so D is multiplied by $2^{k-1}/2^k = 1/2$ hence decreases multiplicatively again, thus showing that the algorithm terminates since D is bounded from below.

When the algorithm terminates, we have for all i, j and k the conditions $B_k \geq (3/4 - \mu_{k,k-1}^2)B_{k-1}$ and $|\mu_{i,j}| \leq 1/2$. If p is the rank of the lattice L, it follows that $n-p$ of the B_i must be equal to zero, and these inequalities show that it must be the *first* $n-p$ B_i, since $B_i = 0$ implies $B_j = 0$ for $j < i$. Since the vector space generated by the $\mathbf{b}_i^*$ for $i \leq n-p$ is the same as the space generated by the $\mathbf{b}_i$ for $i \leq n-p$, it follows that $\mathbf{b}_i = 0$ for $i \leq n-p$. Since the $\mathbf{b}_i$ form a generating set for L over $\mathbb{Z}$ throughout the algorithm, the $\mathbf{b}_i$ for $i > n-p$ also generate L, hence they form a basis since there are exactly p of them, and this basis is LLL reduced by construction. It also follows from the vanishing of the $\mathbf{b}_i$ for $i \leq n-p$ that the first $n-p$ columns H_i of H are relation vectors for our initial $\mathbf{b}_i$. Since H is an integer matrix with determinant ± 1, it is an easy exercise to see that these columns form a basis of the space of relation vectors for the initial $\mathbf{b}_i$ (Exercise 25). □

This algorithm is essentially due to M. Pohst and called by him the MLLL algorithm (for Modified LLL, see [Poh2]).

We leave as an excellent exercise for the reader to write an all-integer version of Algorithm 2.6.8 when the Gram matrix is integral (see Exercise 26).

Summary. We have seen a number of modifications and variations on the basic LLL Algorithm 2.6.3. Most of these can be combined. We summarize them here.

(1) The Gram-Schmidt formulas of step 2 can be modified to use only the Gram matrix of the $\mathbf{b}_i$ (see Remark (2) after Algorithm 2.6.3).
(2) If the Gram-Schmidt matrix is integral, the computation can be done entirely with integers (see Algorithm 2.6.7).
(3) If floating point computations are used, care must be taken during the computation of the $\mathbf{b}_i \cdot \mathbf{b}_j$ and when the nearest integer to a $\mu_{k,l}$ is computed (see Remark (4) after Algorithm 2.6.3).
(4) If we want better quality vectors than those output by the LLL algorithm, we can use deep insertion to improve the output (see Algorithm 2.6.4).
(5) If the vectors $\mathbf{b}_i$ are not linearly independent, we must use Algorithm 2.6.8, combined if desired with any of the preceding variations.

2.7 Applications of the LLL Algorithm

2.7.1 Computing the Integer Kernel and Image of a Matrix

In Section 2.4.3 we have seen how to apply the Hermite normal form algorithms to the computation of the image and kernel of an integer matrix A. It is clear that this can also be done using the MLLL algorithm (in fact its integer version, see Exercise 26). Indeed if we set $\mathbf{b}_j$ to be the columns of A, the vectors $\mathbf{b}_i$ output by Algorithm 2.6.8 form an LLL-reduced basis of the image of A and the relation vectors H_i for $i \leq r = n - p$ form a basis of the integer kernel of A. If desired, the result given by Algorithm 2.6.8 can be improved in two ways. First, the relation vectors H_i for $i \leq r$ are not LLL-reduced, so it is useful to LLL-reduce them to obtain small relations. This means that we will multiply the first r column of H on the right by an $r \times r$ invertible matrix over $\mathbb{Z}$, and this of course leaves H unimodular.

Second, although the basis $\mathbf{b}_i$ for $r < i \leq n$ is already an LLL-reduced basis for the image of A hence cannot be improved much, the last p columns of H (which express the LLL-reduced $\mathbf{b}_i$ in terms of the initial $\mathbf{b}_i$) can be large and in many situations it is desirable to reduce their size. Here we must *not* LLL-reduce these columns since the corresponding image vectors $\mathbf{b}_i$ would not be anymore LLL-reduced in general. (This is of course a special case of the important but difficult problem of simultaneously reducing a lattice basis and its dual, see [Sey2].) We still have some freedom however since we can replace any column H_i for $i > r$ by

$$H_i - \sum_{j \le r} m_j H_j$$

for any $m_j \in \mathbb{Z}$ since this will not change the $\mathbf{b}_i$ and will preserve the relation $\det(H) = \pm 1$. To choose the m_j close to optimally we proceed as follows. Let C be the Gram matrix of the vectors H_j for $j \le r$. Using Algorithm 2.2.1 compute $X = (x_1, \ldots, x_r)^t$ solution to the linear system $CX = V_i$, where V_i is the column vector whose j-th element is equal to $H_i \cdot H_j$ (here the scalar product is the usual one). Then by elementary geometric arguments it is clear that the vector $\sum_{j \le r} x_j H_j$ is the projection of H_i on the real vector space generated by the H_j for $j \le r$, hence a close to optimal choice of the m_j is to choose $m_j = \lfloor x_j \rceil$. Since we have several linear systems to solve using the same matrix, it is preferable to invert the matrix using Algorithm 2.2.2 and this gives the following algorithm.

Algorithm 2.7.1 (Kernel and Image of a Matrix Using LLL). Given an $m \times n$ matrix A with integral entries, this algorithm computes an $n \times n$ matrix H and a number p with the following properties. The matrix H has integral entries and is of determinant equal to ± 1 (i.e. $H \in \mathrm{GL}_n(\mathbb{Z})$). The first $n - p$ columns of H form an LLL-reduced basis of the integer kernel of A. The product of A with the last p columns of H give an LLL-reduced basis of the image of A, and the coefficients of these last p columns are small.

1. [Apply MLLL] Perform Algorithm 2.6.8 on the vectors $\mathbf{b}_i$ equal to the columns of A, the Euclidean scalar product being the usual scalar product on vectors. We thus obtain p and a matrix $H \in \mathrm{GL}_n(\mathbb{Z})$. Set $r \leftarrow n - p$.

2. [LLL-reduce the kernel] Using the integral LLL-Algorithm 2.6.7, replace the first r vectors of H by an LLL-reduced basis of the lattice that they generate.

3. [Compute inverse of Gram matrix] Let C be the Gram matrix of the H_j for $j \le r$ (i.e. $C_{j,k} = H_j \cdot H_k$ for $1 \le j, k \le r$), set $D \leftarrow C^{-1}$ computed using Algorithm 2.2.2, and set $i \leftarrow r$.

4. [Finished?] Set $i \leftarrow i + 1$. If $i > n$, output the matrix H and the number p and terminate the algorithm.

5. [Modify H_i] Let V be the r-dimensional column vector whose j-th coordinate is $H_i \cdot H_j$. Set $X \leftarrow DV$, and for $j \le r$ set $m_j \leftarrow \lfloor x_j \rceil$, where x_j is the j-th component of X. Finally, set $H_i \leftarrow H_i - \sum_{1 \le j \le r} m_j H_j$ and go to step 4.

A practical implementation of this algorithm should use only an all-integer version of Algorithm 2.6.8 (see Exercise 26), and the other steps can be similarly modified so that all the computations are done with integers only.

If only the integer kernel of A is wanted, we may replace the test $B_k < (0.75 - \mu_{k,k-1}^2) B_{k-1}$ by $B_k = 0$, which avoids most of the swaps and gives a much faster algorithm. Since this algorithm is very useful, we give explicitly the complete integer version.

Algorithm 2.7.2 (Kernel over $\mathbb{Z}$ Using LLL). Given an $m \times n$ matrix A with integral entries, this algorithm finds an LLL-reduced $\mathbb{Z}$-basis for the kernel of A. We use an auxiliary $n \times n$ integral matrix H. We denote by H_j the j-th column of H and (to keep notations similar to the other LLL algorithms) by $\mathbf{b}_j$ the j-th column of A. All computations are done using integers only. We use an auxiliary set of flags $f_1, \ldots, f_n$ (which will be such that $f_k = 0$ if and only if $B_k = 0$).

1. [Initialize] Set $k \leftarrow 2$, $k_{\max} \leftarrow 1$, $d_0 \leftarrow 1$, $t \leftarrow \mathbf{b}_1 \cdot \mathbf{b}_1$ and $H \leftarrow I_n$. If $t \neq 0$ set $d_1 \leftarrow t$ and $f_1 \leftarrow 1$, otherwise set $d_1 \leftarrow 1$ and $f_1 \leftarrow 0$.

2. [Incremental Gram-Schmidt] If $k \leq k_{\max}$ go to step 3. Otherwise, set $k_{\max} \leftarrow k$ and for $j = 1, \ldots, k$ (in that order) do as follows. If $f_j = 0$ and $j < k$, set $\lambda_{k,j} \leftarrow 0$. Otherwise, set $u \leftarrow \mathbf{b}_k \cdot \mathbf{b}_j$ and for each $i = 1, \ldots, j-1$ (in that order) such that $f_i \neq 0$ set

$$u \leftarrow \frac{d_i u - \lambda_{k,i}\lambda_{j,i}}{d_{i-1}}$$

   (the result is in $\mathbb{Z}$), then, if $j < k$ set $\lambda_{k,j} \leftarrow u$ and if $j = k$ set $d_k \leftarrow u$ and $f_k \leftarrow 1$ if $u \neq 0$, $d_k \leftarrow d_{k-1}$ and $f_k \leftarrow 0$ if $u = 0$.

3. [Test $f_k = 0$ and $f_{k-1} \neq 0$] If $f_{k-1} \neq 0$, execute Sub-algorithm REDI$(k, k-1)$ above. If $f_{k-1} \neq 0$ and $f_k = 0$, execute Sub-algorithm SWAPK(k) below, set $k \leftarrow \max(2, k-1)$ and go to step 3. Otherwise, for each $l = k-2, k-3, \ldots, 1$ (in this order) such that $f_l \neq 0$, execute Sub-algorithm REDI(k, l) above, then set $k \leftarrow k+1$.

4. [Finished?] If $k \leq n$ go to step 2. Otherwise, let $r+1$ be the least index such that $f_i \neq 0$ ($r = n$ if all f_i are equal to 0). Using Algorithm 2.6.7, output an LLL-reduced basis of the lattice generated by the linearly independent vectors $H_1, \ldots, H_r$ and terminate the algorithm.

Sub-algorithm SWAPK(k). Exchange the vectors H_k and H_{k-1}, and if $k > 2$, for all j such that $1 \leq j \leq k-2$ exchange $\lambda_{k,j}$ with $\lambda_{k-1,j}$. Set $\lambda \leftarrow \lambda_{k,k-1}$. If $\lambda = 0$, set $d_{k-1} \leftarrow d_{k-2}$, exchange f_{k-1} and f_k (i.e. set $f_{k-1} \leftarrow 0$ and $f_k \leftarrow 1$), set $\lambda_{k,k-1} \leftarrow 0$ and for $i = k+1, \ldots, k_{\max}$ set $\lambda_{i,k} \leftarrow \lambda_{i,k-1}$ and $\lambda_{i,k-1} \leftarrow 0$.

If $\lambda \neq 0$, for $i = k+1, \ldots, k_{\max}$ set $\lambda_{i,k-1} \leftarrow \lambda\lambda_{i,k-1}/d_{k-1}$, then set $t \leftarrow d_k$, $d_{k-1} \leftarrow \lambda^2/d_{k-1}$, $d_k \leftarrow d_{k-1}$ then for $j = k+1, \ldots, k_{\max}-1$ and for $i = j+1, \ldots, k_{\max}$ set $\lambda_{i,j} \leftarrow \lambda_{i,j}d_{k-1}/t$ and finally for $j = k+1, \ldots, k_{\max}$ set $d_j \leftarrow d_j d_{k-1}/t$. Terminate the sub-algorithm.

Remarks.

(1) Since $f_i = 0$ implies $\lambda_{k,i} = 0$, time can be saved in a few places by first testing whether f_i vanishes. The proof of the validity of this algorithm is left as an exercise (Exercise 24).

(2) It is an easy exercise to show that in this algorithm

$$d_k = \det\left((\mathbf{b}_i \cdot \mathbf{b}_j)_{1 \leq i,j \leq k, B_i B_j \neq 0}\right)$$

and that $d_j\mu_{i,j} \in \mathbb{Z}$ (see Exercise 29).

(3) An annoying aspect of Algorithm SWAPK is that when $\lambda \neq 0$, in addition to the usual updating, we must also update the quantities d_j and $\lambda_{i,j}$ for all i and j such that $k+1 \leq j < i \leq k_{\max}$. This comes from the single fact that the new value of d_k is different from the old one, and suggests that a suitable modification of the definition of d_k can suppress this additional updating. This is indeed the case (see Exercise 30). Unfortunately, with this modification, it is the reduction algorithm REDI which needs much additional updating. I do not see how to suppress the extra updating in SWAPK and in REDI simultaneously.

2.7.2 Linear and Algebraic Dependence Using LLL

Now let us see how to apply the LLL algorithm to the problem of $\mathbb{Z}$-linear independence. Let $z_1, z_2, \ldots, z_n$ be n complex numbers, and the problem is to find a $\mathbb{Z}$-dependence relation between them, if one exists. Assume first that the z_i are real. For a large number N, consider the positive definite quadratic form in the a_i:

$$Q(\mathbf{a}) = a_2^2 + a_3^2 + \cdots + a_n^2 + N(z_1a_1 + z_2a_2 + \cdots + z_na_n)^2.$$

This form is represented as a sum of n squares of linearly independent linear forms in the a_i, hence defines a Euclidean scalar product on $\mathbb{R}^n$, as long as $z_1 \neq 0$, which we can of course assume. If N is large, a "short" vector of $\mathbb{Z}^n$ for this form will necessarily be such that $|z_1a_1 + \cdots + z_na_n|$ is small, and also the a_i for $i > 1$ not too large. Hence, if the z_i are really $\mathbb{Z}$-linearly dependent, by choosing a suitable constant N the dependence relation (which will make $z_1a_1 + \cdots + z_na_n$ equal to 0 up to roundoff errors) will be discovered. The choice of the constant N is subtle, and depends in part on what one knows about the problem. If the $|z_i|$ are not too far from 1 (meaning between 10^{-6} and 10^6, say), and are known with an absolute (or relative) precision ϵ, then one should take N between $1/\epsilon$ and $1/\epsilon^2$, but ϵ should also be taken quite small: if one expects the coefficients a_i to be of the order of a, then one might take $\epsilon = a^{-1.5n}$, but in any case $\epsilon < a^{-n}$.

Hence, we will start with the $\mathbf{b}_i$ being the standard basis of $\mathbb{Z}^n$, and use LLL with the quadratic form above. One nice thing is that step 2 of the LLL algorithm can be avoided completely. Indeed, one has the following lemma.

Lemma 2.7.3. *With the above notations, if we execute the complete Gram-Schmidt orthogonalization procedure on the standard basis of $\mathbb{Z}^n$ and the quadratic form*

$$Q(\mathbf{a}) = a_2^2 + a_3^2 + \cdots + a_n^2 + N(z_1a_1 + z_2a_2 + \cdots + z_na_n)^2$$

we have $\mu_{i,1} = z_i/z_1$ for $2 \leq i \leq n$, $\mu_{i,j} = 0$ if $2 \leq j < i \leq n$, $\mathbf{b}_i^* = \mathbf{b}_i - (z_i/z_1)\mathbf{b}_1$, $B_1 = Nz_1^2$, and $B_k = 1$ for $2 \leq k \leq n$.

The proof is trivial by induction.

It is easy to modify these ideas to obtain an algorithm which also works for complex numbers z_i. In this case, the quadratic form that we can take is

$$Q(\mathbf{a}) = a_3^2 + \cdots + a_n^2 + N|z_1a_1 + z_2a_2 + \cdots + z_na_n|^2,$$

since the expression which multiplies N is now a sum of *two* squares of linear forms, and these forms will be independent if and only if z_1/z_2 is not real. We can however always satisfy this condition by a suitable reordering: if there exists i and j such that $z_i/z_j \notin \mathbb{R}$, then by applying a suitable permutation of the z_i, we may assume that $z_1/z_2 \notin \mathbb{R}$. On the other hand, if $z_i/z_j \in \mathbb{R}$ for all i and j, then we can apply the algorithm to the real numbers $1, z_2/z_1, \ldots, z_n/z_1$.

All this leads to the following algorithm.

Algorithm 2.7.4 (Linear Dependence). Given n complex numbers $z_1, \ldots, z_n$, (as approximations), a large number N chosen as explained above, this algorithm finds $\mathbb{Z}$-linear combinations of small modulus between the z_i. We assume that all the z_i are non-zero, and that if one of the ratios z_i/z_j is not real, the z_i are reordered so that the ratio z_2/z_1 is not real.

1. [Initialize] Set $\mathbf{b}_i \leftarrow [0, \ldots, 1, \ldots, 0]^t$, i.e. as a column vector the i^{th} element of the standard basis of $\mathbb{Z}^n$. Then, set $\mu_{i,j} \leftarrow 0$ for all i and j with $3 \le j < i \le n$, $B_1 \leftarrow |z_1|^2$, $B_2 \leftarrow \mathrm{Im}(z_1\bar{z}_2)$, $B_k \leftarrow 1$ for $3 \le k \le n$, $\mu_{i,1} \leftarrow \mathrm{Re}(z_1\bar{z}_i)/B_1$ for $2 \le i \le n$.

   Now if $B_2 \neq 0$ (i.e. if we are in the complex case), do the following: set $\mu_{i,2} \leftarrow \mathrm{Im}(z_1\bar{z}_i)/B_2$ for $3 \le i \le n$, $B_2 \leftarrow N \cdot B_2^2/B_1$. Otherwise (in the real case), set $\mu_{i,2} \leftarrow 0$ for $3 \le i \le n$, $B_2 \leftarrow 1$.

2. [Execute LLL] Set $B_1 \leftarrow NB_1$, $k \leftarrow 2$, $k_{\max} \leftarrow n$, $H \leftarrow I_n$ and go to step 3 of the LLL Algorithm 2.6.3.

3. [Terminate] Output the coefficients $\mathbf{b}_i$ as coefficients of linear combinations of the z_i with small modulus, the best one being probably $\mathbf{b}_1$.

Implementation advice. Algorithm 2.7.4 performs slightly better if z_1 is the number with the largest modulus. Hence one should try to reorder the z_i so that this is the case. (Note that it may not be possible to do so, since if the z_i are not all real, one must have z_2/z_1 non-real.)

Remarks.

(1) The reason why the first component plays a special role comes from the choice of the quadratic form. To be more symmetrical, one could choose instead

$$Q(\mathbf{a}) = a_1^2 + a_2^2 + a_3^2 + \cdots + a_n^2 + N|z_1a_1 + z_2a_2 + \cdots + z_na_n|^2$$

both in the real and complex case. The result would be more symmetrical in the variables a_i, but then we cannot avoid executing step 2 of the LLL

algorithm, i.e. the Gram-Schmidt reduction procedure, which in practice can take a non-negligible proportion of the running time. Hence the above non-symmetric version (due to W. Neumann) is probably better.

(2) We can express the linear dependence algorithm in terms of matrices instead of quadratic forms as follows (for simplicity we use the symmetrical version and we assume the z_i real). Set $S = \sqrt{N}$. We must then find the LLL reduction of the following $(n+1) \times n$ matrix:

$$\begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 1 \\ Sz_1 & Sz_2 & \dots & Sz_n \end{pmatrix}.$$

(3) We have not used at all the multiplicative structure of the field $\mathbb{C}$. This means that essentially the same algorithm can be used to find linear dependencies between elements of a k-dimensional vector space over $\mathbb{R}$ for any k. This essentially reduces to the MLLL algorithm, except that thanks to the number N we can better handle imprecise vectors.

(4) A different method for finding linear dependence relations based on an algorithm which is a little different from the LLL algorithm, is explained and analyzed in detail in [HJLS]. It is not clear which should be preferred.

A special case of Algorithm 2.7.4 is when $z_i = \alpha^{i-1}$, where α is a given complex number. Then finding a $\mathbb{Z}$-linear relation between the z_i is equivalent to finding a polynomial $A \in \mathbb{Z}[X]$ such that $A(\alpha) = 0$, i.e. an algebraic relation for α. This is very useful in practice. (From the implementation advice given above we should choose $z_i = \alpha^{n-i}$ instead if $\alpha > 1$.)

In this case however, some modifications may be useful. First note that Lemma 2.7.3 stays essentially the same if we replace the quadratic form $Q(\mathbf{a})$ by

$$Q(\mathbf{a}) = \lambda_2 a_2^2 + \lambda_3 a_3^2 + \cdots + \lambda_n a_n^2 + N|z_1 a_1 + z_2 a_2 + \cdots + z_n a_n|^2$$

where the λ_i are arbitrary positive real numbers (see Exercise 32). Now when testing for algebraic relations, we may or may not know in advance the degree of the relation. Assume that we do. (For example, if $\alpha = \sqrt{2} + \sqrt{3} + \sqrt{5}$ we know that the relation will be of degree 8.) Then (choosing $z_i = \alpha^{n-i}$) we would like to have small coefficients for α^{n-i} with i small, and allow larger ones for i large. This amounts to choosing λ_i large for small i, and small for large i. One choice could be $\lambda_i = A^{n-i}$ for some reasonable constant $A > 1$ (at least such that A^n is much smaller than N). In other words, we look for an algebraic relation for z_i/A.

In other situations, we do not know in advance the degree of the relation, or even if the number is algebraic or not. In this case, it is probably not necessary to modify Algorithm 2.7.4, i.e. we simply choose $\lambda_i = 1$ for all i.

2.7.3 Finding Small Vectors in Lattices

For many applications, even though the LLL algorithm does not always give us the smallest vector in a lattice, the vectors which are obtained are sufficiently reasonable to give good results. We have seen one such example in the preceding section, where LLL was used to find linear dependence relations between real or complex numbers. In some cases, however, it is absolutely necessary to find one of the smallest vectors in a lattice, or more generally all vectors having norm less than or equal to some constant. This problem is hard, and in a slightly modified form is known to be NP-complete, i.e. equivalent to the most difficult reasonable problems in computer science for which no polynomial time algorithm is known. (For a thorough discussion of NP-completeness and related matters, see for example [AHU].) Nonetheless, we must give an algorithm to solve it, keeping in mind that any algorithm will probably be exponential time with respect to the dimension.

Using well known linear algebra algorithms (over $\mathbb{R}$ and not over $\mathbb{Z}$), we can assume that the matrix defining the Euclidean inner product on $\mathbb{R}^n$ is diagonal with respect to the canonical basis, say $Q(\mathbf{x}) = q_{1,1}x_1^2 + q_{2,2}x_2^2 + \cdots + q_{n,n}x_n^2$. If we want $Q(\mathbf{x}) \leq C$, say, then we must choose $|x_1| \leq \sqrt{C/q_{1,1}}$. Once x_1 is chosen, we choose $|x_2| \leq \sqrt{(C - q_{1,1}x_1^2)/q_{2,2}}$, and so on. This leads to n nested loops, and in addition it is desirable to have n variable and not fixed. Hence it is not as straightforward to implement as it may seem. The idea is to use implicitly a lexicographic ordering of the vectors $\mathbf{x}$. If we generalize this to non-diagonal quadratic forms, this leads to the following algorithm.

Algorithm 2.7.5 (Short Vectors). If Q is a positive definite quadratic form given by

$$Q(\mathbf{x}) = \sum_{i=1}^{n} q_{i,i}\left(x_i + \sum_{j=i+1}^{n} q_{i,j}x_j\right)^2$$

and a positive constant C, this algorithm outputs all the non-zero vectors $\mathbf{x} \in \mathbb{Z}^n$ such that $Q(\mathbf{x}) \leq C$, as well as the value of $Q(\mathbf{x})$. Only one of the two vectors in the pair $(\mathbf{x}, -\mathbf{x})$ is actually given.

1. [Initialize] Set $i \leftarrow n$, $T_i \leftarrow C$, $U_i \leftarrow 0$.
2. [Compute bounds] Set $Z \leftarrow \sqrt{T_i/q_{i,i}}$, $L_i \leftarrow \lfloor Z - U_i \rfloor$, $x_i \leftarrow \lceil -Z - U_i \rceil - 1$.
3. [Main loop] Set $x_i \leftarrow x_i + 1$. If $x_i > L_i$, set $i \leftarrow i + 1$ and go to step 3. Otherwise, if $i > 1$, set $T_{i-1} \leftarrow T_i - q_{i,i}(x_i + U_i)^2$, $i \leftarrow i - 1$, $U_i \leftarrow \sum_{j=i+1}^{n} q_{i,j}x_j$, and go to step 2.
4. [Solution found] If $\mathbf{x} = \mathbf{0}$, terminate the algorithm, otherwise output $\mathbf{x}$, $Q(\mathbf{x}) = C - T_1 + q_{1,1}(x_1 + U_1)^2$ and go to step 3.

Now, although this algorithm (due in this form to Fincke and Pohst) is quite efficient in small dimensions, it is far from being the whole story. Since

we have at our disposal the LLL algorithm which is efficient for finding short vectors in a lattice, we can use it to modify our quadratic form so as to shorten the length of the search. More precisely, let $R = (r_{i,j})$ be the upper triangular matrix defined by $r_{i,i} = \sqrt{q_{i,i}}$, $r_{i,j} = r_{i,i}q_{i,j}$ for $1 \le i < j \le n$, $r_{i,j} = 0$ for $1 \le j < i \le n$. Then

$$Q(\mathbf{x}) = \mathbf{x}^t R^t R \mathbf{x}.$$

Now call $\mathbf{r}_i$ the columns of R and $\mathbf{r}'_i$ the rows of R^{-1}. Then from the identity $R^{-1}R\mathbf{x} = \mathbf{x}$ we obtain $x_i = \mathbf{r}'_i R\mathbf{x}$, hence by the Cauchy-Schwarz inequality,

$$x_i^2 \le \|\mathbf{r}'_i\|^2(\mathbf{x}^t R^t R\mathbf{x}) \le \|\mathbf{r}'_i\|^2 C.$$

This bound is quite sharp since for example when the quadratic form is diagonal, we have $\|\mathbf{r}'_i\|^2 = 1/q_{i,i}$ and the bound that we obtain for x_1, say, is as usual $\sqrt{C/q_{1,1}}$. Using the LLL algorithm on the rows of R^{-1}, however, will in general drastically reduce the norms of these rows, and hence improve correspondingly the search for short vectors.

As a final improvement, we note that the implicit lexicographic ordering on the vectors $\mathbf{x}$ used in Algorithm 2.7.5 is not unique, and in particular we can permute the coordinates as we like. This adds some more freedom on our reduction of the matrix R. Before giving the final algorithm, due to Fincke and Pohst, we give the standard method to obtain the so-called Cholesky decomposition of a positive definite quadratic form, i.e. to obtain Q in the form used in Algorithm 2.7.5.

Algorithm 2.7.6 (Cholesky Decomposition). Let A be a real symmetric matrix of order n defining a positive definite quadratic form Q. This algorithm computes constants $q_{i,j}$ and a matrix R such that

$$Q(\mathbf{x}) = \sum_{i=1}^{n} q_{i,i}\left(x_i + \sum_{j=i+1}^{n} q_{i,j}x_j\right)^2$$

or equivalently in matrix form $A = R^t R$.

1. [Initialize] For all i and j such that $1 \le i \le j \le n$ set $q_{i,j} \leftarrow a_{i,j}$, then set $i \leftarrow 0$.
2. [Loop on i] Set $i \leftarrow i+1$. If $i = n$, go to step 4. Otherwise, for $j = i+1, \ldots, n$ set $q_{j,i} \leftarrow q_{i,j}$ and $q_{i,j} \leftarrow q_{i,j}/q_{i,i}$.
3. [Main loop] For all k and l such that $i+1 \le k \le l \le n$ set

$$q_{k,l} \leftarrow q_{k,l} - q_{k,i}q_{i,l}$$

   and go to step 2.
4. [Find matrix R] For $i = 1, \ldots, n$ set $r_{i,i} \leftarrow \sqrt{q_{i,i}}$, then set $r_{i,j} = 0$ if $1 \le j < i \le n$ and $r_{i,j} = r_{i,i}q_{i,j}$ if $1 \le i < j \le n$ and terminate the algorithm.

Note that this algorithm is essentially a reformulation of the Gram-Schmidt orthogonalization procedure in the case where only the Gram matrix is known. (See Proposition 2.5.7 and Remark (2) after Algorithm 2.6.3.)

We can now give the algorithm of Fincke-Pohst for finding vectors of small norm ([Fin-Poh]).

Algorithm 2.7.7 (Fincke-Pohst). Let A be a real symmetric matrix of order n defining a positive definite quadratic form Q, and C be a positive constant. This algorithm outputs all non-zero vectors $\mathbf{x} \in \mathbb{Z}^n$ such that $Q(\mathbf{x}) \leq C$ and the corresponding values of $Q(\mathbf{x})$. As in Algorithm 2.7.5, only one of the two vectors $(\mathbf{x}, -\mathbf{x})$ is actually given.

1. [Cholesky] Apply the Cholesky decomposition Algorithm 2.7.6 to the matrix A, thus obtaining an upper triangular matrix R. Compute also R^{-1} (note that this is easy since R is triangular).

2. [LLL reduction] Apply the LLL algorithm to the n vectors formed by the *rows* of R^{-1}, thus obtaining a unimodular matrix U and a matrix S^{-1} such that $S^{-1} = U^{-1}R^{-1}$. Compute also $S = RU$. (Note that U will simply be the inverse transpose of the matrix H obtained in Algorithm 2.6.3, and this can be directly obtained instead of H in that algorithm, in other words it is not necessary to compute a matrix inverse).

3. [Reorder the columns of S] Call $\mathbf{s}_i$ the columns of S and $\mathbf{s}'_i$ the rows of S^{-1}. Find a permutation σ on $[1, \ldots, n]$ such that

$$\|\mathbf{s}'_{\sigma(1)}\| \geq \|\mathbf{s}'_{\sigma(2)}\| \geq \cdots \geq \|\mathbf{s}'_{\sigma(n)}\|.$$

   Then permute the *columns* of S using the same permutation σ, i.e. replace S by the matrix whose i^{th} column is $\mathbf{s}_{\sigma(i)}$ for $1 \leq i \leq n$.

4. Compute $A_1 \leftarrow S^t S$, and find the coefficients $q_{i,j}$ of the Cholesky decomposition of A_1 using the first three steps of Algorithm 2.7.6 (it is not necessary to compute the new matrix R).

5. Using Algorithm 2.7.5 on the quadratic form Q_1 defined by the symmetric matrix A_1, compute all the non-zero vectors $\mathbf{y}$ such that $Q_1(\mathbf{y}) \leq C$, and for each such vector output $\mathbf{x} = U(y_{\sigma^{-1}(1)}, \ldots, y_{\sigma^{-1}(n)})^t$ and $Q(\mathbf{x}) = Q_1(\mathbf{y})$.

Although this algorithm is still exponential time, and is more complex than Algorithm 2.7.5, in theory and in practice it is much better and should be used systematically except if n is very small (less than 5, say).

Remark. If we want not only small vectors but *minimal* non-zero vectors, the Fincke-Pohst algorithm should be used as follows. First, use the LLL algorithm on the lattice $(\mathbb{Z}^n, Q)$. This will give small vectors in this lattice, and then choose as constant C the smallest norm among the vectors found by LLL, then apply Algorithm 2.7.7.

2.8 Exercises for Chapter 2

1. Prove that if K is a field, any invertible matrix over K is equal to a product of matrices corresponding to elementary column operations. Is this still true if K is not a field, for example for $\mathbb{Z}$?
2. Let $MX = B$ be a square linear system with coefficients in the ring $\mathbb{Z}/p^r\mathbb{Z}$ for some prime number p and some integer $r > 1$. Show how to use Algorithm 2.2.1 over the field $\mathbb{Q}_p$ to obtain at least one solution to the system, if such a solution exists. Compute in particular the necessary p-adic precision.
3. Write an algorithm which decomposes a square matrix M in the form $M = LUP$ as mentioned in the text, where P is a permutation matrix, and L and U are lower and upper triangular matrices respectively (see [AHU] or [PFTV] if you need help).
4. Give a detailed proof of Proposition 2.2.5.
5. Using the notation of Proposition 2.2.5, show that for $k+1 \leq i, j \leq n$, the coefficient $a_{i,j}^{(k)}$ is equal to the $(k+1) \times (k+1)$ minor of M_0 obtained by taking the first k rows and the i-th row, and the first k columns and the j-th column of M_0.
6. Generalize the Gauss-Bareiss method for computing determinants, to the computation of the inverse of a matrix with integer coefficients, and more generally to the other algorithms of this chapter which use elimination.
7. Is it possible to modify the Hessenberg Algorithm 2.2.9 so that when the matrix M has coefficients in $\mathbb{Z}$ all (or most) operations are done on integers and not on rational numbers? (I do not know the answer to this question.)
8. Prove the validity of Algorithm 2.3.1.
9. Prove the validity of Algorithm 2.3.6.
10. Write an algorithm for computing one element of the inverse image, analogous to Algorithm 2.3.4 but using elimination directly instead of using Algorithm 2.3.1, and compare the asymptotic speed with that of Algorithm 2.3.4.
11. Prove the validity of Algorithm 2.3.11 and the uniqueness statement of Proposition 2.3.10.
12. In Algorithm 2.3.9, show that if the columns of M and M' are linearly independent then so are the columns of M_2.
13. Assuming Theorem 2.4.1 (1), prove parts (2) and (3). Also, try and prove (1).
14. Prove the uniqueness part of Theorem 2.4.3.
15. Show that among all possible pairs (u, v) such that $au + bv = d = \gcd(a, b)$, there exists exactly one such that $-|a|/d < v\,\mathrm{sign}(b) \leq 0$, and that in addition we will also have $1 \leq u\,\mathrm{sign}(a) \leq |b|/d$.
16. Generalize Algorithm 2.4.14 to the case where the $n \times n$ square matrix A is not assumed to be non-singular.
17. Let $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ be a 2×2 matrix with integral coefficients such that $ad - bc \neq 0$. If we set $d_2 = \gcd(a, b, c, d)$ and $d_1 = (ad - bc)/d_2$ show directly that there

exists two matrices U and V in $\mathrm{GL}_2(\mathbb{Z})$ such that $A = V \begin{pmatrix} d_1 & 0 \\ 0 & d_2 \end{pmatrix} U$ (this is the special case $n = 2$ of Theorem 2.4.12).

18. Let G be a finite $\mathbb{Z}$-module, hence isomorphic to a quotient L'/L, and let A be a matrix giving the coordinates of some $\mathbb{Z}$-basis of L on some $\mathbb{Z}$-basis of L'. Show that the absolute value of $\det(A)$ is equal to the cardinality of G.

19. Let B be an invertible matrix with real coefficients. Show that there exist matrices K_1, K_2 and A such that $B = K_1 A K_2$, where A is a diagonal matrix with positive diagonal coefficients, and K_1 and K_2 are orthogonal matrices (this is called the *Cartan decomposition* of B). What extra condition can be added so that the decomposition is unique?

20. Prove Proposition 2.5.3 using only matrix-theoretical tools (hint: the matrix Q is diagonalizable since it is real symmetric).

21. Give recursive formulas for the computation of the Gram-Schmidt coefficients $\mu_{i,j}$ and B_i when only the Gram matrix $(\mathbf{b}_i \cdot \mathbf{b}_j)$ is known.

22. Assume that the vector $\mathbf{b}_i$ is replaced by some other vector $\mathbf{b}_k$ in the Gram-Schmidt process. Compute the new value of $B_i = \mathbf{b}_i^* \cdot \mathbf{b}_i^*$ in terms of the $\mu_{k,j}$ and B_j for $j < i$.

23. Prove Theorem 2.6.2 (5) and the validity of the LLL Algorithm 2.6.3.

24. Prove that the formulas of Algorithm 2.6.3 become those of Algorithm 2.6.7 when we set $\lambda_{i,j} \leftarrow d_j \mu_{i,j}$ and $d_i \leftarrow d_{i-1} B_i$.

25. Show that at the end of Algorithm 2.6.8 the first $n-p$ columns H_i of the matrix H form a basis of the space of relation vectors for the initial $\mathbf{b}_i$.

26. Write an all integer version of Algorithm 2.6.8, generalizing Algorithm 2.6.7 to not necessarily independent vectors. The case corresponding to $B_k = 0$ but $\mu_{k,k-1} \neq 0$ must be treated with special care.

27. (This is not really an exercise, just food for thought). Generalize to modules over principal ideal domains R the results and algorithms given about lattices. For example, generalize the LLL algorithm to the case where R is either the ring of integers of a number field (see Chapter 4) assumed to be principal, or is the ring $K[X]$ where $K = \mathbb{Q}$, $K = \mathbb{R}$ or $K = \mathbb{C}$. What can be said when $K = \mathbb{F}_p$? Give applications to the problem of linear or algebraic dependence of power series.

28. Compare the performance of Algorithms 2.7.2 and 2.4.10 (in the author's implementations, Algorithm 2.7.2 is by far superior).

29. Prove that the quantities that occur in Algorithm 2.7.2 are indeed all integral. In particular, show that $d_k = \det(\mathbf{b}_i \cdot \mathbf{b}_j)_{1 \leq i,j \leq k, B_i B_j \neq 0}$ and that $d_j \mu_{i,j} \in \mathbb{Z}$.

30. Set by convention $\mu_{k,0} = 1$, $\mu_{k,k} = B_k$, $j(k) = \max\{j,\ 0 \leq j \leq k, \mu_{k,j} \neq 0\}$, $d_k = \prod_{1 \leq i \leq k} \mu_{i,j(i)}$ and $\lambda_{k,j} = d_j \mu_{k,j}$ for $k > j$.

    a) Modify Sub-algorithm SWAPK so that it uses this new definition of d_k and $\lambda_{k,j}$. In other words, find the formulas giving the new values of the d_j, f_j and $\lambda_{k,j}$ in terms of the old ones after exchanging $\mathbf{b}_k$ and $\mathbf{b}_{k-1}$. In particular show that, contrary to Sub-algorithm SWAPK, d_k is always unchanged.

    b) Modify also Sub-algorithm REDI accordingly. (Warning: d_k may be modified, hence all d_j and $\lambda_{i,j}$ for $i > j > k$.)

    c) Show that we still have $d_j \in \mathbb{Z}$ and $\lambda_{k,j} \in \mathbb{Z}$ (this is much more difficult

and is analogous to the integrality property of the Gauss-Bareiss Algorithm 2.2.6 and the sub-resultant Algorithm 3.3.1 that we will study in Chapter 3).

31. It can be proved that $s_k = \sum_{n \geq 1} (n(n+1)\cdots(n+k-1))^{-3}$ is of the form $a\pi^2 + b$ where a and b are rational numbers when k is even, and also when k is odd if the middle coefficient $(n + (k-1)/2)$ is only raised to the power -2 instead of -3. Compute s_k for $k \leq 4$ using Algorithm 2.7.4.

32. Prove Lemma 2.7.3 and its generalization mentioned after Algorithm 2.7.4. Write the corresponding algebraic dependence algorithm.

33. Let U be a non-singular real square matrix of order n, and let Q be the positive definite quadratic form defined by the real symmetric matrix $U^t U$. Using explicitly the inverse matrix V of U, generalize Algorithm 2.7.5 to find small values of Q on $\mathbb{Z}^n$ (Algorithm 2.7.5 corresponds to the case where U is a triangular matrix). Hint: if you have trouble, see [Knu2] Section 3.3.4.C.

Chapter 3

Algorithms on Polynomials

Excellent book references on this subject are [Knu2] and [GCL].

3.1 Basic Algorithms

3.1.1 Representation of Polynomials

Before studying algorithms on polynomials, we need to decide how they will be represented in an actual program. The straightforward way is to represent a polynomial

$$P(X) = a_n X^n + a_{n-1} X^{n-1} + \cdots + a_1 X + a_0$$

by an array $a[0]$, $a[1]$, ..., $a[n]$. The only difference between different implementations is that the array of coefficients can also be written in reverse order, with $a[0]$ being the coefficient of X^n. We will always use the first representation. Note that the leading coefficient a_n may be equal to 0, although usually this will not be the case.

The true degree of the polynomial P will be denoted by $\deg(P)$, and the coefficient of $X^{\deg(P)}$, called the leading coefficient of P, will be denoted by $\ell(P)$. In the example above, if, as is usually the case, $a_n \neq 0$, then $\deg(P) = n$ and $\ell(P) = a_n$.

The coefficients a_i may belong to any commutative ring with unit, but for many algorithms it will be necessary to specify the base ring. If this base ring is itself a ring of polynomials, we are then dealing with polynomials in several variables, and the representation given above (called the dense representation) is very inefficient, since multivariate polynomials usually have very few non-zero coefficients. In this situation, it is better to use the so-called *sparse* representation, where only the exponents and coefficients of the non-zero monomials are stored. The study of algorithms based on this kind of representation would however carry us too far afield, and will not be considered here. In any case, practically all the algorithms that we will need use only polynomials in one variable.

The operations of addition, subtraction and multiplication by a scalar, i.e. the vector space operations, are completely straightforward and need not be discussed. On the other hand, it is necessary to be more specific concerning multiplication and division.

3.1.2 Multiplication of Polynomials

As far as multiplication is concerned, one can of course use the straightforward method based on the formula:

$$\left(\sum_{i=0}^{m} a_i X^i\right)\left(\sum_{j=0}^{n} b_j X^j\right) = \sum_{k=0}^{n+m} c_k X^k,$$

where

$$c_k = \sum_{i=0}^{k} a_i b_{k-i},$$

where it is understood that $a_i = 0$ if $i > m$ and $b_j = 0$ if $j > n$. This method requires $(m+1)(n+1)$ multiplications and mn additions. Since in general multiplications are much slower than additions, especially if the coefficients are multi-precision numbers, it is reasonable to count only the multiplication time. If $T(M)$ is the time for multiplication of elements in the base ring, the running time is thus $O(mnT(M))$. It is possible to multiply polynomials faster than this, however. We will not study this in detail, but will give an example. Assume we want to multiply two polynomials of degree 1. The straightforward method above gives:

$$(a_1 X + a_0)(b_1 X + b_0) = c_2 X^2 + c_1 X + c_0,$$

with

$$c_0 = a_0 b_0, \quad c_1 = a_0 b_1 + a_1 b_0, \quad c_2 = a_1 b_1.$$

As mentioned, this requires 4 multiplications and 1 addition. Consider instead the following alternate method for computing the c_k:

$$c_0 = a_0 b_0, \qquad c_2 = a_1 b_1,$$

$$d = (a_1 - a_0)(b_1 - b_0), \qquad c_1 = c_0 + (c_2 - d).$$

This requires only 3 multiplications, but 4 additions (subtraction and addition times are considered identical). Hence it is faster if one multiplication in the base ring is slower than 3 additions. This is almost always the case, especially if the base ring is not too simple or involves large integers. Furthermore, this method can be used for any degree, by recursively splitting the polynomials in two pieces of approximately equal degrees.

There is a generalization of the above method which is based on Lagrange's interpolation formula. To compute $A(X)B(X)$, which is a polynomial of degree $m+n$, compute its value at $m+n+1$ suitably chosen points. This involves only $m+n+1$ multiplications. One can then recover the coefficients of $A(X)B(X)$ (at least if the ring has characteristic zero) by using a suitable algorithmic form of Lagrange's interpolation formula. The overhead which this implies is unfortunately quite large, and for practical implementations, the reader is advised either to stick to the straightforward method, or to use the recursive splitting procedure mentioned above.

3.1.3 Division of Polynomials

We assume here that the polynomials involved have coefficients in a field K, (or at least that all the divisions which occur make sense. Note that if the coefficients belong to an integral domain, one can extend the scalars and assume that they in fact belong to the quotient field). The ring $K[X]$ is then a Euclidean domain, and this means that given two polynomials A and B with $B \neq 0$, there exist unique polynomials Q and R such that

$$A = BQ + R, \quad \text{with } \deg(R) < \deg(B)$$

(where as usual we set $\deg(0) = -\infty$). As we will see in the next section, this means that most of the algorithms described in Chapter 1 for the Euclidean domain $\mathbb{Z}$ can be applied here as well.

First however we must describe algorithms for computing Q and R. The straightforward method can easily be implemented as follows. For a non-zero polynomial Z, recall that $\ell(Z)$ is the leading coefficient of Z. Then:

Algorithm 3.1.1 (Euclidean Division). Given two polynomials A and B in $K[X]$ with $B \neq 0$, this algorithm finds Q and R such that $A = BQ + R$ and $\deg(R) < \deg(B)$.

1. [Initialize] Set $R \leftarrow A$, $Q \leftarrow 0$.
2. [Finished?] If $\deg(R) < \deg(B)$ then terminate the algorithm.
3. [Find coefficient] Set

$$S \leftarrow \frac{\ell(R)}{\ell(B)} X^{\deg(R)-\deg(B)},$$

then $Q \leftarrow Q + S$, $R \leftarrow R - S \cdot B$ and go to step 2.

Note that the multiplication $S \cdot B$ in step 3 is not really a polynomial multiplication, but simply a scalar multiplication followed by a shift of coefficients. Also, if division is much slower than multiplication, it is worthwhile to compute only once the inverse of $\ell(B)$, so as to have only multiplications in step 3. The running time of this algorithm is hence

$$O(\deg(B)(\deg(Q) + 1)T(M)),$$

(of course, $\deg(Q) = \deg(A) - \deg(B)$ if $\deg(A) \geq \deg(B)$).

Remark. The subtraction $R \leftarrow R - S \cdot B$ in step 3 of the algorithm must be carefully written: by definition of S, the coefficient of $X^{\deg R}$ must become exactly zero, so that the degree of R decreases. If however the base field is for example $\mathbb{R}$ or $\mathbb{C}$, the elements of K will only be represented with finite precision, and in general the operation $\ell(R) - \ell(B)(\ell(R)/\ell(B))$ will not give

exactly zero but a very small number. Hence it is absolutely necessary to set it exactly equal to zero when implementing the algorithm.

Note that the assumption that K is a field is not strictly necessary. Since the only divisions which take place in the algorithm are divisions by the leading coefficient of B, it is sufficient to assume that this coefficient is invertible in K, as for example is the case if B is monic. We will see an example of this in Algorithm 3.5.5 below (see also Exercise 3).

The abstract value $T(M)$ does not reflect correctly the computational complexity of the situation. In the case of multiplication, the abstract $T(M)$ used made reasonable sense. For example, if the base ring K was $\mathbb{Z}$, then $T(M)$ would be the time needed to multiply two integers whose size was bounded by the coefficients of the polynomials A and B. On the contrary, in Algorithm 3.1.1 the coefficients explode, as can easily be seen, hence this abstract measure of complexity $T(M)$ does not make sense, at least in $\mathbb{Z}$ or $\mathbb{Q}$. On the other hand, in a field like $\mathbb{F}_p$, $T(M)$ does make sense.

Now these theoretical considerations are in fact very important in practice: Among the most used base fields (or rings), there can be no coefficient explosion in $\mathbb{F}_p$ (or more generally any finite field), or in $\mathbb{R}$ or $\mathbb{C}$ (since in that case the coefficients are represented as limited precision quantities). On the other hand, in the most important case of $\mathbb{Q}$ or $\mathbb{Z}$, such an explosion does take place, and one must be ready to deal with it.

There is however one other important special case where no explosion takes place, that is when B is a monic polynomial ($\ell(B) = 1$), and A and B are in $\mathbb{Z}[X]$. In this case, there is no division in step 3 of the algorithm.

In the general case, one can avoid divisions by multiplying the polynomial A by $\ell(B)^{\deg(A)-\deg(B)+1}$. This gives an algorithm which is not really more efficient than Algorithm 3.1.1, but which is neater and will be used in the next section. Knuth calls it "pseudo-division" of polynomials. It is as follows:

Algorithm 3.1.2 (Pseudo-Division). Let K be a ring, A and B be two polynomials in $K[X]$ with $B \neq 0$, and set $m \leftarrow \deg(A)$, $n \leftarrow \deg(B)$, $d \leftarrow \ell(B)$. Assume that $m \geq n$. This algorithm finds Q and R such that $d^{m-n+1}A = BQ+R$ and $\deg(R) < \deg(B)$.

1. [Initialize] Set $R \leftarrow A$, $Q \leftarrow 0$, $e \leftarrow m - n + 1$.
2. [Finished?] If $\deg(R) < \deg(B)$ then set $q \leftarrow d^e$, $Q \leftarrow qQ$, $R \leftarrow qR$ and terminate the algorithm.
3. [Find coefficient] Set

$$S \leftarrow \ell(R)X^{\deg(R)-\deg(B)},$$

then $Q \leftarrow d \cdot Q + S$, $R \leftarrow d \cdot R - S \cdot B$, $e \leftarrow e - 1$ and go to step 2.

Since the algorithm does not use any division, we assume only that K is a ring, for example one can have $K = \mathbb{Z}$. Note also that the final multiplication by $q = d^e$ is needed only to get the exact power of d, and this is necessary for

some applications such as the sub-resultant algorithm (see 3.3). If it is only necessary to get some constant multiple of Q and R, one can dispense with e and q entirely.

3.2 Euclid's Algorithms for Polynomials

3.2.1 Polynomials over a Field

Euclid's algorithms given in Section 1.3 can be applied with essentially no modification to polynomials with coefficients in a field K where no coefficient explosion takes place (such as $\mathbb{F}_p$). In fact, these algorithms are even simpler, since it is not necessary to have special versions à la Lehmer for multi-precision numbers. They are thus as follows:

Algorithm 3.2.1 (Polynomial GCD). Given two polynomials A and B over a field K, this algorithm determines their GCD in $K[X]$.

1. [Finished?] If $B = 0$, then output A as the answer and terminate the algorithm.
2. [Euclidean step] Let $A = B \cdot Q + R$ with $\deg(R) < \deg(B)$ be the Euclidean division of A by B. Set $A \leftarrow B$, $B \leftarrow R$ and go to step 1.

The extended version is the following:

Algorithm 3.2.2 (Extended Polynomial GCD). Given two polynomials A and B over a field K, this algorithm determines (U, V, D) such that $AU + BV = D = (A, B)$.

1. [Initialize] Set $U \leftarrow 1$, $D \leftarrow A$, $V_1 \leftarrow 0$, $V_3 \leftarrow B$.
2. [Finished?] If $V_3 = 0$ then let $V \leftarrow (D - AU)/B$ (the division being exact), output (U, V, D) and terminate the algorithm.
3. [Euclidean step] Let $D = QV_3 + R$ be the Euclidean division of D by V_3. Set $T \leftarrow U - V_1 Q$, $U \leftarrow V_1$, $D \leftarrow V_3$, $V_1 \leftarrow T$, $V_3 \leftarrow R$ and go to step 2.

Note that the polynomials U and V given by this algorithm are polynomials of the smallest degree, i.e. they satisfy $\deg(U) < \deg(B/D)$, $\deg(V) < \deg(A/D)$.

If the base field is $\mathbb{R}$ or $\mathbb{C}$, then the condition $B = 0$ of Algorithm 3.2.1 (or $V_3 = 0$ in Algorithm 3.2.2) becomes meaningless since numbers are represented only approximately. In fact, polynomial GCD's over these fields, although mathematically well defined, cannot be used in practice since the coefficients are only approximate. Even if we assume the coefficients to be given by some formula which allows us to compute them as precisely as we desire, the computation cannot usually be done. Consider for example the computation of

$$\gcd(X - \pi, X^2 - 6\zeta(2)),$$

where $\zeta(s) = \sum_{n \geq 1} n^{-s}$ is the Riemann zeta function. Although we can compute the coefficients to as many decimal places as we desire, algebra alone will not tell us that this GCD is equal to $X - \pi$ since $\zeta(2) = \pi^2/6$. The point of this discussion is that one should keep in mind that it is meaningless in practice to compute polynomial GCD's over $\mathbb{R}$ or $\mathbb{C}$.

On the other hand, if the base field is $\mathbb{Q}$, the above algorithms make perfect sense. Here, as already mentioned for Euclidean division, the practical problem of the coefficient explosion will occur, and since several divisions are performed, it will be much worse.

To be specific, if p is small, the GCD of two polynomials of $\mathbb{F}_p[X]$ of degree 1000 can be computed in a reasonable amount of time, say a few seconds, while the GCD of polynomials in $\mathbb{Q}[X]$ (even with very small integer coefficients) could take incredibly long, years maybe, because of coefficient explosion. Hence in this case it is absolutely necessary to use better algorithms. We will see this in Sections 3.3 and 3.6.1. Before that, we need some important results about polynomials over a Unique Factorization Domain (UFD).

3.2.2 Unique Factorization Domains (UFD's)

Definition 3.2.3. *Let $\mathcal{R}$ be an integral domain (i.e. a commutative ring with unit 1 and no zero divisors). We say that $u \in \mathcal{R}$ is a* unit *if u has a multiplicative inverse in $\mathcal{R}$. If a and b are elements of $\mathcal{R}$ with $b \neq 0$, we say that b* divides *a (and write $b \mid a$) if there exists $q \in \mathcal{R}$ such that $a = bq$. Since $\mathcal{R}$ is an integral domain, such a q is unique and denoted by a/b. Finally $p \in \mathcal{R}$ is called an* irreducible element *or a* prime element *if q divides p implies that either q or p/q is a unit.*

Definition 3.2.4. *A ring $\mathcal{R}$ is called a unique factorization domain (UFD) if $\mathcal{R}$ is an integral domain, and if every non-unit $x \in \mathcal{R}$ can be written in the form $x = \prod p_i$, where the p_i are (not necessarily distinct) prime elements, and if this form is unique up to permutation and multiplication of the primes by units.*

Important examples of UFD's are given by the following theorem (see [Kap], [Sam]):

Theorem 3.2.5.

(1) *If $\mathcal{R}$ is a principal ideal domain (i.e. $\mathcal{R}$ is an integral domain and every ideal is principal), then $\mathcal{R}$ is a UFD. In particular, Euclidean domains (i.e. those having a Euclidean division) are UFD's.*

(2) *If $\mathcal{R}$ is the ring of algebraic integers of a number field (see Chapter 4), then $\mathcal{R}$ is a UFD if and only if $\mathcal{R}$ is a principal ideal domain.*
(3) *If $\mathcal{R}$ is a UFD, then the polynomial rings $\mathcal{R}[X_1, \ldots, X_n]$ are also UFD's.*

Note that the converse of (1) is not true in general: for example the ring $\mathbb{C}[X, Y]$ is a UFD (by (3)), but is not a principal ideal domain (the ideal generated by X and Y is not principal).

We will not prove Theorem 3.2.5 (see Exercise 6 for a proof of (3)), but we will prove some basic lemmas on UFD's before continuing further.

Theorem 3.2.6. *Let $\mathcal{R}$ be a UFD. Then*

(1) *If p is prime, then for all a and b in $\mathcal{R}$, $p \mid ab$ if and only if $p \mid a$ or $p \mid b$.*
(2) *If $a \mid bc$ and a has no common divisor with b other than units, then $a \mid c$.*
(3) *If a and b have no common divisor other than units, then if a and b divide $c \in \mathcal{R}$, then $ab \mid c$.*
(4) *Given a set $S \subset \mathcal{R}$ of elements of $\mathcal{R}$, there exists $d \in \mathcal{R}$ called a greatest common divisor (GCD) of the elements of S, and having the following properties: d divides all the elements of S, and if e is any element of $\mathcal{R}$ dividing all the elements of S, then $e \mid d$. Furthermore, if d and d' are two GCD's of S, then d/d' is a unit.*

Proof. (1) Assume $p \mid ab$. Since $\mathcal{R}$ is a UFD, one can write $a = \prod_{1 \le i \le m} p_i$ and $b = \prod_{m+1 \le i \le m+n} p_i$, the p_i being not necessarily distinct prime elements of $\mathcal{R}$. On the other hand, since $ab/p \in \mathcal{R}$ we can also write $ab = p \prod_j q_j$ with prime elements q_j. By the uniqueness of prime decomposition, since $ab = \prod_{1 \le i \le m+n} p_i$ we deduce that p is equal to a unit times one of the p_i. Hence, if $i \le m$, then $p \mid a$, while if $i > m$, then $p \mid b$, proving (1).

(2) We prove (2) by induction on the number n of prime factors of b, counted with multiplicity. If $n = 0$ then b is a unit and $a \mid c$. Assume the result true for $n-1$, and let $bc = qa$ with $n \ge 1$. Let p be a prime divisor of b. p divides qa, and by assumption p does not divide a. Hence by (1) p divides q, and we can write $b'c = q'a$ with $b' = b/p$, $q' = q/p$. Since b' has only $n-1$ prime divisors, (2) follows by induction.

(3) Write $c = qa$ with $q \in \mathcal{R}$. Since $b \mid c$, by (2) we deduce that $b \mid q$, hence $ab \mid c$.

(4) For every element $s \in S$, write

$$s = u \prod_p p^{v_p(s)},$$

where u is a unit, the product is over *all* distinct prime elements of $\mathcal{R}$ up to units, and $v_p(s)$ is the number of times that the prime p occurs in s, hence is 0 for all but finitely many p. Set

$$d = \prod_p p^{\alpha_p}, \qquad \text{where } \alpha_p = \min_{s \in S} v_p(s).$$

This min is of course equal to 0 for all but a finite number of p, and it is clear that d satisfies the conditions of the theorem. □

We will say that the elements of S are *coprime* if their GCD is a unit. By definition of a UFD, this is equivalent to saying that no prime element is a common divisor. Note that if $\mathcal{R}$ is not only a UFD but also a principal ideal domain (for example when the UFD $\mathcal{R}$ is the ring of algebraic integers in a number field), then the coprimality condition is equivalent to saying that the ideal generated by the elements is the whole ring $\mathcal{R}$. This is however *not* true in general. For example, in the UFD $\mathbb{C}[X, Y]$, the elements X and Y are coprime, but the ideal which they generate is the set of polynomials P such that $P(0, 0) = 0$, and this is not the whole ring.

3.2.3 Polynomials over Unique Factorization Domains

Definition 3.2.7. *Let $\mathcal{R}$ be a UFD, and $A \in R[X]$. We define the* content *of A and write* $\operatorname{cont}(A)$ *as a GCD of the coefficients of A. We say that A is* primitive *if* $\operatorname{cont}(A)$ *is a unit, i.e. if its coefficients are coprime. Finally, if $A \neq 0$ the polynomial $A/\operatorname{cont}(A)$ is primitive, and is called the primitive part of A, and denoted* $\operatorname{pp}(A)$ *(in the case $A = 0$ we define* $\operatorname{cont}(A) = 0$, $\operatorname{pp}(A) = 0$*).*

The fundamental result on these notions, due to Gauss, is as follows:

Theorem 3.2.8. *Let A and B be two polynomials over a UFD $\mathcal{R}$. Then there exists a unit $u \in \mathcal{R}$ such that*

$$\operatorname{cont}(A \cdot B) = u \operatorname{cont}(A) \operatorname{cont}(B), \qquad \operatorname{pp}(A \cdot B) = u^{-1} \operatorname{pp}(A) \operatorname{pp}(B).$$

In particular, the product of two primitive polynomials is primitive.

Proof. Since $A = \operatorname{cont}(A) \operatorname{pp}(A)$, it is clear that this theorem is equivalent to the statement that the product of two primitive polynomials A and B is primitive. Assume the contrary. Then there exists a prime $p \in \mathcal{R}$ which divides all the coefficients of AB. Write $A(X) = \sum a_i X^i$ and $B(X) = \sum b_i X^i$. By assumption there exists a j such that a_j is not divisible by p, and similarly a k such that b_k is not divisible by p. Choose j and k as small as possible. The coefficient of X^{j+k} in AB is $a_j b_k + a_{j+1} b_{k-1} + \cdots + a_{j+k} b_0 + a_{j-1} b_{k+1} + \cdots + a_0 b_{k+j}$, and all the terms in this sum are divisible by p except the term $a_j b_k$ (since j and k have been chosen as small as possible), and $a_j b_k$ itself is not divisible by p since p is prime. Hence p does not divide the coefficient of X^{j+k} in AB, contrary to our assumption, and this proves the theorem. □

Corollary 3.2.9. *Let A and B be two polynomials over a UFD $\mathcal{R}$. Then there exists units u and v in $\mathcal{R}$ such that*

$$\operatorname{cont}(\gcd(A,B)) = u \gcd(\operatorname{cont}(A), \operatorname{cont}(B)),$$

$$\operatorname{pp}(\gcd(A,B)) = v \gcd(\operatorname{pp}(A), \operatorname{pp}(B)).$$

3.2.4 Euclid's Algorithm for Polynomials over a UFD

We can now give Euclid's algorithm for polynomials defined over a UFD. The important point to notice is that the sequence of operations will be essentially identical to the corresponding algorithm over the quotient field of the UFD, but the algorithm will run much faster. This is because implementing arithmetic in the quotient field (say in $\mathbb{Q}$ if $R = \mathbb{Z}$) will involve taking GCD's in the UFD all the time, many more than are needed to execute Euclid's algorithm. Hence the following algorithm is always to be preferred to Algorithm 3.2.1 when the coefficients of the polynomials are in a UFD. We will however study in the next section a more subtle and efficient method.

Algorithm 3.2.10 (Primitive Polynomial GCD). Given two polynomials A and B with coefficients in a UFD $\mathcal{R}$, this algorithm computes a GCD of A and B, using only operations in $\mathcal{R}$. We assume that we already have at our disposal algorithms for (exact) division and for GCD in $\mathcal{R}$.

1. [Reduce to primitive] If $B = 0$, output A and terminate. Otherwise, set $a \leftarrow \operatorname{cont}(A)$, $b \leftarrow \operatorname{cont}(B)$, $d \leftarrow \gcd(a,b)$, $A \leftarrow A/a$, $B \leftarrow B/b$.
2. [Pseudo division] Compute R such that $\ell(B)^{\deg(A)-\deg(B)+1}A = BQ + R$ using Algorithm 3.1.2. If $R = 0$ go to step 4. If $\deg(R) = 0$, set $B \leftarrow 1$ and go to step 4.
3. [Replace] Set $A \leftarrow B$, $B \leftarrow \operatorname{pp}(R) = R/\operatorname{cont}(R)$ and go to step 2.
4. [Terminate] Output $d \cdot B$ and terminate the algorithm.

In the next section, we will see an algorithm which is in general faster than the above algorithm. There are also other methods which are often even faster, but are based on quite different ideas. Consider the case where $R = \mathbb{Z}$. Instead of trying to control the explosion of coefficients, we simply put ourselves in a field where this does not occur, i.e. in the finite field $\mathbb{F}_p$ for suitable primes p. If one finds that the GCD modulo p has degree 0 (and this will happen often), then if p is suitably chosen it will follow that the initial polynomials are coprime over $\mathbb{Z}$. Even if the GCD is not of degree 0, it is in general quite easy to deduce from it the GCD over $\mathbb{Z}$. We will come back to this question in Section 3.6.1.

3.3 The Sub-Resultant Algorithm

3.3.1 Description of the Algorithm

The main inconvenience of Algorithm 3.2.10 is that we compute the content of R in step 3 each time, and this is a time consuming operation. If we did not reduce R at all, then the coefficient explosion would make the algorithm much slower, and this is also not acceptable. There is a nice algorithm due to Collins, which is a good compromise and which is in general faster than Algorithm 3.2.10, although the coefficients are larger. The idea is that one can give an a priori divisor of the content of R, which is sufficiently large to replace the content itself in the reduction. This algorithm is derived from the algorithm used to compute the resultant of two polynomials (see Section 3.3.2), and is called the sub-resultant algorithm. We could still divide A and B by their content from time to time (say every 10 iterations), but this would be a very bad idea (see Exercise 4).

Algorithm 3.3.1 (Sub-Resultant GCD). Given two polynomials A and B with coefficients in a UFD $\mathcal{R}$, this algorithm computes a GCD of A and B, using only operations in $\mathcal{R}$. We assume that we already have at our disposal algorithms for (exact) division and for GCD in $\mathcal{R}$.

1. [Initializations and reductions] If $\deg(B) > \deg(A)$ exchange A and B. Now if $B = 0$, output A and terminate the algorithm, otherwise, set $a \leftarrow \mathrm{cont}(A)$, $b \leftarrow \mathrm{cont}(B)$, $d \leftarrow \gcd(a, b)$, $A \leftarrow A/a$, $B \leftarrow B/b$, $g \leftarrow 1$ and $h \leftarrow 1$.
2. [Pseudo division] Set $\delta \leftarrow \deg(A) - \deg(B)$. Using Algorithm 3.1.2, compute R such that $\ell(B)^{\delta+1}A = BQ + R$. If $R = 0$ go to step 4. If $\deg(R) = 0$, set $B \leftarrow 1$ and go to step 4.
3. [Reduce remainder] Set $A \leftarrow B$, $B \leftarrow R/(gh^{\delta})$, $g \leftarrow \ell(A)$, $h \leftarrow h^{1-\delta}g^{\delta}$ and go to step 2. (Note that all the divisions which may occur in this step give a result in the ring $\mathcal{R}$.)
4. [Terminate] Output $d \cdot B/\mathrm{cont}(B)$ and terminate the algorithm.

It is not necessary for us to give the proof of the validity of this algorithm, since it is long and is nicely done in [Knu2]. The main points to notice are as follows: first, it is clear that this algorithm gives exactly the same sequence of polynomials as the straightforward algorithm, but multiplied or divided by some constants. Consequently, the only thing to prove is that all the quantities occurring in the algorithm stay in the ring $\mathcal{R}$. This is done by showing that all the coefficients of the intermediate polynomials as well as the quantities h are determinants of matrices whose coefficients are coefficients of A and B, hence are in the ring $\mathcal{R}$.

Another result which one obtains in proving the validity of the algorithm is that in the case $R = \mathbb{Z}$, if $m = \deg(A)$, $n = \deg(B)$, and N is an upper bound for the absolute value of the coefficients of A and B, then the coefficients of the intermediate polynomials are all bounded by the quantity

$$N^{m+n}(m+1)^{n/2}(n+1)^{m/2},$$

and this is reasonably small. One can then show that the execution time for computing the GCD of two polynomials of degree n over $\mathbb{Z}$ when their coefficients are bounded by N in absolute value is $O(n^4(\ln Nn)^2)$.

I leave as an exercise to the reader the task of writing an extended version of Algorithm 3.3.1 which gives polynomials U and V such that $AU + BV = r(A,B)$, where $r \in \mathcal{R}$. All the operations must of course be done in $\mathcal{R}$ (see Exercise 5). Note that it is not always possible to have $r = 1$. For example, if $A(X) = X$ and $B(X) = 2$, then $(A,B) = 1$ but for any U and V the constant term of $AU + BV$ is even.

3.3.2 Resultants and Discriminants

Let A and B be two polynomials over an integral domain $\mathcal{R}$ with quotient field K, and let $\overline{K}$ be an algebraic closure of K.

Definition 3.3.2. *Let $A(X) = a(X-\alpha_1)\cdots(X-\alpha_m)$ and $B(X) = b(X-\beta_1)\cdots(X-\beta_n)$ be the decomposition of A and B in $\overline{K}$. Then the* resultant *$R(A,B)$ of A and B is given by one of the equivalent formulas:*

$$\begin{aligned} R(A,B) &= a^n B(\alpha_1)\cdots B(\alpha_m) \\ &= (-1)^{mn} b^m A(\beta_1)\cdots A(\beta_n) \\ &= a^n b^m \prod_{1\le i\le m, 1\le j\le n} (\alpha_i - \beta_j). \end{aligned}$$

Definition 3.3.3. *If $A \in \mathcal{R}[X]$, with $m = \deg(A)$, the* discriminant $\operatorname{disc}(A)$ *of A is equal to the expression:*

$$(-1)^{m(m-1)/2} R(A,A')/\ell(A),$$

where A' is the derivative of A.

The main point about these definitions is that resultants and discriminants have coefficients in $\mathcal{R}$. Indeed, by the symmetry in the roots α_i, it is clear that the resultant is a function of the symmetric functions of the roots, hence is in K. It is not difficult to see that the coefficient a^n insures that $R(A,B) \in \mathcal{R}$. Another way to see this is to prove the following lemma.

Lemma 3.3.4. *If $A(X) = \sum_{0\le i\le m} a_i X^i$ and $B(X) = \sum_{0\le i\le n} b_i X^i$, then the resultant $R(A,B)$ is equal to the determinant of the following $(n+m)\times(n+m)$ matrix:*

$$\begin{pmatrix}
a_m & a_{m-1} & a_{m-2} & \dots & a_1 & a_0 & 0 & 0 & \dots & 0 \\
0 & a_m & a_{m-1} & a_{m-2} & \dots & a_1 & a_0 & 0 & \dots & 0 \\
0 & 0 & a_m & a_{m-1} & a_{m-2} & \dots & a_1 & a_0 & \dots & 0 \\
\vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
0 & 0 & \dots & 0 & a_m & a_{m-1} & a_{m-2} & \dots & a_1 & a_0 \\
b_n & b_{n-1} & \dots & b_2 & b_1 & b_0 & 0 & 0 & \dots & 0 \\
0 & b_n & b_{n-1} & \dots & b_2 & b_1 & b_0 & 0 & \dots & 0 \\
0 & 0 & b_n & b_{n-1} & \dots & b_2 & b_1 & b_0 & \dots & 0 \\
\vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\
0 & 0 & \dots & 0 & b_n & b_{n-1} & \dots & b_2 & b_1 & b_0
\end{pmatrix},$$

where the coefficients of A are repeated on $n = \deg(B)$ rows, and the coefficients of B are repeated on $m = \deg(A)$ rows.

The above matrix is called Sylvester's matrix. Since the only non-zero coefficients of the first column of this matrix are a_m and b_n, it is clear that $R(A,B)$ is not only in $\mathcal{R}$ but in fact divisible (in $\mathcal{R}$) by $\gcd(\ell(A),\ell(B))$. In particular, if $B = A'$, $R(A,A')$ is divisible by $\ell(A)$, hence $\operatorname{disc}(A)$ is also in $\mathcal{R}$.

Proof. Call M the above matrix. Assume first that the α_i and β_j are all distinct. Consider the $(n+m)\times(n+m)$ Vandermonde matrix $V = (v_{i,j})$ defined by $v_{i,j} = \beta_j^{m+n-i}$ if $j \le n$, $v_{i,j} = \alpha_{j-n}^{m+n-i}$ if $n+1 \le j \le n+m$. Then the Vandermonde determinant $\det(V)$ is non-zero since we assumed the α_i and β_j distinct, and we have

$$\det(V) = \prod_{i<j}(\beta_i - \beta_j)\prod_{i<j}(\alpha_i - \alpha_j)\prod_{i,j}(\beta_i - \alpha_j).$$

On the other hand, it is clear that

$$MV = \begin{pmatrix}
\beta_1^{n-1}A(\beta_1) & \dots & \beta_n^{n-1}A(\beta_n) & 0 & \dots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
A(\beta_1) & \dots & A(\beta_n) & 0 & \dots & 0 \\
0 & \dots & 0 & \alpha_1^{m-1}B(\alpha_1) & \dots & \alpha_m^{m-1}B(\alpha_m) \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & \dots & 0 & B(\alpha_1) & \dots & B(\alpha_m)
\end{pmatrix},$$

hence $\det(MV)$ is equal to the product of the two diagonal block determinants, which are again Vandermonde determinants. Hence we obtain:

$$\det(MV) = A(\beta_1)\cdots A(\beta_n)B(\alpha_1)\cdots B(\alpha_m)\prod_{i<j}(\beta_i - \beta_j)\prod_{i<j}(\alpha_i - \alpha_j).$$

Comparing with the formula for $\det(V)$ and using $\det(V) \neq 0$ we obtain

$$\det(M)\prod_{i,j}(\beta_i-\alpha_j)=A(\beta_1)\cdots A(\beta_n)B(\alpha_1)\cdots B(\alpha_m).$$

Since clearly $A(\beta_1)\cdots A(\beta_n)=a^n\prod_{i,j}(\beta_i-\alpha_j)$, the lemma follows in the case where all the α_j and β_i are distinct, and it follows in general by a continuity argument or by taking the roots as formal variables. □

Note that by definition, the resultant of A and B is equal to 0 if and only if A and B have a common root, hence if and only if $\deg(A,B)>0$. In particular, the discriminant of a polynomial A is zero if and only if A has a non-trivial square factor, hence if and only if $\deg(A,A')>0$.

The definition of the discriminant that we have given may seem a little artificial. It is motivated by the following proposition.

Proposition 3.3.5. *Let $A\in R[X]$ with $m=\deg(A)$, and let α_i be the roots of A in $\overline{K}$. Then we have*

$$\operatorname{disc}(A)=\ell(A)^{m-1+\deg(A')}\prod_{1\le i<j\le m}(\alpha_i-\alpha_j)^2.$$

Proof. If A has multiple roots, both sides are 0. So we assume that A has only simple roots. Now if $a=\ell(A)$, we have

$$A'(X)=a\sum_i\prod_{j\neq i}(X-\alpha_j)$$

hence

$$A'(\alpha_i)=a\prod_{j\neq i}(\alpha_i-\alpha_j).$$

Thus we obtain

$$R(A,A')=a^{m+\deg(A')}(-1)^{m(m-1)/2}\prod_{i<j}(\alpha_i-\alpha_j)^2$$

thus proving the proposition. Note that we have $\deg(A')=m-1$, except when the characteristic of R is non-zero and divides m. □

The following corollary follows immediately from the definitions.

Corollary 3.3.6. *We have $R(A_1A_2,A_3)=R(A_1,A_3)R(A_2,A_3)$ and*

$$\operatorname{disc}(A_1A_2)=\operatorname{disc}(A_1)\operatorname{disc}(A_2)(R(A_1,A_2))^2.$$

Resultants and discriminants will be fundamental in our handling of algebraic numbers. Now the nice fact is that we have already done essentially all the work necessary to compute them: a slight modification of Algorithm 3.3.1 will give us the resultant of A and B.

Algorithm 3.3.7 (Sub-Resultant). Given two polynomials A and B with coefficients in a UFD $\mathcal{R}$, this algorithm computes the resultant of A and B.

1. [Initializations and reductions] If $A = 0$ or $B = 0$, output 0 and terminate the algorithm. Otherwise, set $a \leftarrow \mathrm{cont}(A)$, $b \leftarrow \mathrm{cont}(B)$, $A \leftarrow A/a$, $B \leftarrow B/b$, $g \leftarrow 1$, $h \leftarrow 1$, $s \leftarrow 1$ and $t \leftarrow a^{\deg(B)} b^{\deg(A)}$. Finally, if $\deg(A) < \deg(B)$ exchange A and B and if in addition $\deg(A)$ and $\deg(B)$ are odd set $s \leftarrow -1$.
2. [Pseudo division] Set $\delta \leftarrow \deg(A) - \deg(B)$. If $\deg(A)$ and $\deg(B)$ are odd, set $s \leftarrow -s$. Finally, compute R such that $\ell(B)^{\delta+1} A = BQ + R$ using Algorithm 3.1.2.
3. [Reduce remainder] Set $A \leftarrow B$ and $B \leftarrow R/(gh^{\delta})$.
4. [Finished?] Set $g \leftarrow \ell(A)$, $h \leftarrow h^{1-\delta} g^{\delta}$. If $\deg(B) > 0$ go to step 2, otherwise set $h \leftarrow h^{1-\deg(A)} \ell(B)^{\deg(A)}$ output $s \cdot t \cdot h$ and terminate the algorithm.

Proof. Set $A_0 = A$, $A_1 = B$, let A_i be the sequence of polynomials generated by this algorithm, and let R_i be the remainders obtained in step 2. Let t be the index such that $\deg(A_{t+1}) = 0$. Set $d_k = \deg(A_k)$, $\ell_k = \ell(A_k)$, and let g_k and h_k be the quantities g and h in stage k, so that $g_0 = h_0 = 1$. Finally set $\delta_k = d_k - d_{k+1}$. Denoting by β_i the roots of A_k, we clearly have for $k \geq 1$:

$$\begin{aligned}
R(A_{k-1}, A_k) &= (-1)^{d_{k-1} d_k} \ell_k^{d_{k-1}} \prod_{1 \leq i \leq d_k} A_{k-1}(\beta_i) \\
&= (-1)^{d_{k-1} d_k} \ell_k^{d_{k-1}} \prod_{1 \leq i \leq d_k} \frac{R_{k+1}(\beta_i)}{\ell_k^{\delta_{k-1}+1}} \\
&= (-1)^{d_{k-1} d_k} \ell_k^{d_{k-1} - d_k(\delta_{k-1}+1)} \prod_{1 \leq i \leq d_k} R_{k+1}(\beta_i) \\
&= (-1)^{d_{k-1} d_k} \ell_k^{d_{k-1} - d_k(\delta_{k-1}+1) - d_{k+1}} R(A_k, g_{k-1} h_{k-1}^{\delta_{k-1}} A_{k+1}).
\end{aligned}$$

Now using $R(A, cB) = c^{\deg(A)} R(A, B)$ and the identities $g_k = \ell_k$ and $h_k = h_{k-1}^{1-\delta_{k-1}} g_k^{\delta_{k-1}}$ for $k \geq 1$, we see that the expression simplifies to

$$R(A_{k-1}, A_k) = (-1)^{d_{k-1} d_k} \frac{g_{k-1}^{d_k} h_{k-1}^{d_{k-1}-1}}{g_k^{d_{k+1}} h_k^{d_k - 1}} R(A_k, A_{k+1}).$$

Using $d_{t+1} = 0$, hence $\delta_t = d_t$, we finally obtain

$$\begin{aligned}
R(A, B) &= (-1)^{\sum_{1 \leq k \leq t} d_{k-1} d_k} h_t^{1-\delta_t} R(A_t, A_{t+1}) \\
&= (-1)^{\sum_{1 \leq k \leq t} d_{k-1} d_k} h_t^{1-d_t} \ell_{t+1}^{d_t} \\
&= (-1)^{\sum_{1 \leq k \leq t} d_{k-1} d_k} h_{t+1},
\end{aligned}$$

thus proving the validity of the algorithm. □

Note that it is the same kind of argument and simplifications which show that the A_k have coefficients in the same ring $\mathcal{R}$ as the coefficients of A and B, and that the h_k also belong to $\mathcal{R}$. In fact, we have just proved for instance that $h_{t+1} \in \mathcal{R}$.

Finally, to compute discriminants of polynomials, one simply uses Algorithm 3.3.7 and the formula

$$\operatorname{disc}(A) = (-1)^{m(m-1)/2} R(A, A')/\ell(A),$$

where $m = \deg(A)$.

3.3.3 Resultants over a Non-Exact Domain

Although resultants and GCD's are similar, from the computational point of view, there is one respect in which they completely differ. It does make practical sense to compute (approximate) resultants over $\mathbb{R}$, $\mathbb{C}$ or $\mathbb{Q}_p$, while it does not make sense for GCD's as we have already explained. When dealing with resultants of polynomials with such non-exact coefficients we must however be careful *not* to use the sub-resultant algorithm. For one thing, it is tailored to avoid denominator explosion when the coefficients are, for example, rational numbers or rational functions in other variables. But most importantly, it would simply give wrong results, since the remainders R obtained in the algorithm are only approximate; hence a zero leading coefficient could appear as a very small non-zero number, leading to havoc in the next iteration.

Hence, in this case, the natural solution is to evaluate directly Sylvester's determinant. Now the usual Gaussian elimination method for computing determinants also involves dividing by elements of the ring to which the coefficients belong. In the case of the ring $\mathbb{Z}$, say, this is not a problem since the quotient of two integers will be represented exactly as a rational number. Even for non-exact rings like $\mathbb{R}$, the quotient is another real number given to a slightly worse and computable approximation. On the other hand, in the case where the coefficients are themselves polynomials in another variable over some non-exact ring like $\mathbb{R}$, although one could argue in the same way using rational functions, the final result will not in general simplify to a polynomial as it should, for the same reason as before.

To work around this problem, we must use the Gauss-Bareiss Algorithm 2.2.6 which has exactly the property of keeping all the computations in the initial base ring. Keep in mind, as already mentioned after Algorithm 2.2.6, that if some division of elements of $\mathbb{R}[X]$ (say) is required, then Euclidean division must be used, i.e. we *must* get a polynomial as a result.

Hence to compute resultants we can apply this algorithm to Sylvester's matrix, even when the coefficients are not exact. (In the case of exact coefficients, this algorithm will evidently also work, but will be slower than the

sub-resultant algorithm.) Since Sylvester's matrix is an $(n+m) \times (n+m)$ matrix, it is important to note that simple row operations can reduce it to an $n \times n$ matrix to which we can then apply the Gauss-Bareiss algorithm (see Exercise 8).

Remark. The Gauss-Bareiss method and the sub-resultant algorithm are in fact closely linked. It is possible to adapt the sub-resultant algorithm so as to give correct answers in the non-exact cases that we have mentioned (see Exercise 10), but the approach using determinants is probably safer.

3.4 Factorization of Polynomials Modulo p

3.4.1 General Strategy

We now consider the problem of factoring polynomials. In practice, for polynomials in one variable the most important base rings are $\mathbb{Z}$ (or $\mathbb{Q}$), $\mathbb{F}_p$ or $\mathbb{Q}_p$. Factoring over $\mathbb{R}$ or $\mathbb{C}$ is equivalent to root finding, hence belongs to the domain of numerical analysis. We will give a simple but efficient method for this in Section 3.6.3.

Most factorization methods rely on factorization methods over $\mathbb{F}_p$, hence we will consider this first. In Section 1.6, we have given algorithms for finding roots of polynomials modulo p, and explained that no polynomial-time deterministic algorithm is known to do this (if one does not assume the GRH). The more general case of factoring is similar. The algorithms that we will describe are probabilistic, but are quite efficient.

Contrary to the case of polynomials over $\mathbb{Z}$, polynomials over $\mathbb{F}_p$ have a tendency to have several factors. Hence the problem is not only to break up the polynomial into two pieces (at least), but to factor completely the polynomial as a product of powers of irreducible (i.e. prime in $\mathcal{R}[X]$) polynomials. This is done in four steps, in the following way.

Algorithm 3.4.1 (Factor in $\mathbb{F}_p[X]$). Let $A \in \mathbb{F}_p[X]$ be monic (since we are over a field, this does not restrict the generality). This algorithm factors A as a product of powers of irreducible polynomials in $\mathbb{F}_p[X]$.

1. [Squarefree factorization] Find polynomials $A_1, A_2, \ldots, A_k$ in $\mathbb{F}_p[X]$ such that

(1) $A = A_1^1 A_2^2 \cdots A_k^k$,
(2) The A_i are squarefree and coprime.

(This decomposition of A will be called the squarefree factorization of A).

2. [Distinct degree factorization] For $i = 1, \ldots, k$ find polynomials $A_{i,d} \in \mathbb{F}_p[X]$ such that $A_{i,d}$ is the product of all irreducible factors of A_i of degree d (hence $A_i = \prod_d A_{i,d}$).

3. [Final splittings] For each i and d, factor $A_{i,d}$ into $\deg(A_{i,d})/d$ irreducible factors of degree d.
4. [Cleanup] Group together all the identical factors found, order them by degree, output the complete factorization and terminate the algorithm.

Of course, this is only the skeleton of an algorithm since steps 1, 2 and 3 are algorithms by themselves. We will consider them in turn.

3.4.2 Squarefree Factorization

Let $\overline{\mathbb{F}}_p$ be an algebraic closure of $\mathbb{F}_p$. If $A \in \mathbb{F}_p[X]$ is monic, define $A_i(X) = \prod_j (X - \alpha_j)$ where the α_j are the roots of A in $\overline{\mathbb{F}}_p$ of multiplicity exactly equal to i. Since the Galois group of $\overline{\mathbb{F}}_p/\mathbb{F}_p$ preserves the multiplicity of the roots of A, it permutes the α_j, so all the A_i have in fact coefficients in $\mathbb{F}_p$ (this will also follow from the next algorithm). It is clear that they satisfy the conditions of step 1. It remains to give an algorithm to compute them.

If $A = \prod_i A_i^i$ with A_i squarefree and coprime, then $A' = \sum_i \prod_{j \neq i} A_j^j \cdot iA_i' A_i^{i-1}$. Hence, if $T = \gcd(A, A')$, then for all irreducible P dividing T, the exponent $v_P(T)$ of P in the prime decomposition of T can be obtained as follows: P dividing A must divide an A_m for some m. Hence, for all $i \neq m$ in the sum for A', the v_P of the i^{th} summand is greater than or equal to m and for $i = m$ is equal to $m - 1$ if $p \nmid m$, and otherwise the summand is 0 (note that since A_m is squarefree, A_m' cannot be divisible by P). Hence, we obtain that $v_P(T) = m - 1$ if $p \nmid m$, and $v_P(T) \geq m$, so $v_P(T) = m$ (since T divides A) if $p \mid m$. Finally, we obtain the formula

$$T = (A, A') = \prod_{p \nmid i} A_i^{i-1} \prod_{p \mid i} A_i^i.$$

Note that we could have given a much simpler proof over $\mathbb{Z}$, and in that case the exponent of A_i would be equal to $i - 1$ for all i.

Now we define two sequences of polynomials by induction as follows. Set $T_1 = T$ and $V_1 = A/T = \prod_{p \nmid i} A_i$. For $k \geq 1$, set $V_{k+1} = (T_k, V_k)$ if $p \nmid k$, $V_{k+1} = V_k$ if $p \mid k$, and $T_{k+1} = T_k/V_{k+1}$. It is easy to check by induction that

$$V_k = \prod_{i \geq k,\ p \nmid i} A_i \quad \text{and} \quad T_k = \prod_{i > k,\ p \nmid i} A_i^{i-k} \prod_{p \mid i} A_i^i.$$

From this it follows that $A_k = V_k/V_{k+1}$ for $p \nmid k$. We thus obtain all the A_k for $p \nmid k$, and we continue as long as V_k is a non-constant polynomial. When V_k is constant, we have $T_{k-1} = \prod_{p \mid i} A_i^i$ hence there exists a polynomial U such that $T_{k-1}(X) = U^p(X) = U(X^p)$, and this polynomial can be trivially obtained from T_{k-1}. We then start again recursively the whole algorithm of squarefree decomposition on the polynomial U. Transforming the recursive step into a loop we obtain the following algorithm.

Algorithm 3.4.2 (Squarefree Factorization). Let $A \in \mathbb{F}_p[X]$ be a monic polynomial and let $A = \prod_{i \geq 1} A_i^i$ be its squarefree factorization, where the A_i are squarefree and pairwise coprime. This algorithm computes the polynomials A_i, and outputs the pairs (i, A_i) for the values of i for which A_i is not constant.

1. [Initialize] Set $e \leftarrow 1$ and $T_0 \leftarrow A$.
2. [Initialize e-loop] If T_0 is constant, terminate the algorithm. Otherwise, set $T \leftarrow (T_0, T_0')$, $V \leftarrow T_0/T$ and $k \leftarrow 0$.
3. [Finished e-loop?] If V is constant, T must be of the form $T(X) = \sum_{p|j} t_j X^j$, so set $T_0 \leftarrow \sum_{p|j} t_j X^{j/p}$, $e \leftarrow pe$ and go to step 2.
4. [Special case] Set $k \leftarrow k+1$. If $p \mid k$ set $T \leftarrow T/V$ and $k \leftarrow k+1$.
5. [Compute A_{ek}] Set $W \leftarrow (T, V)$, $A_{ek} \leftarrow V/W$, $V \leftarrow W$ and $T \leftarrow T/V$. If A_{ek} is not constant output (ek, A_{ek}). Go to step 3.

3.4.3 Distinct Degree Factorization

We can now assume that we have a *squarefree* polynomial A and we want to group factors of A of the same degree d. This procedure is known as distinct degree factorization and is quite simple. We first need to recall some results about finite fields. Let $P \in \mathbb{F}_p[X]$ be an irreducible polynomial of degree d. Then the field $K = \mathbb{F}_p[X]/P(X)\mathbb{F}_p[X]$ is a finite field with p^d elements. Hence, every element x of the multiplicative group K^* satisfies the equation $x^{p^d-1} = 1$, therefore every element of K satisfies $x^{p^d} = x$. This shows that P is a divisor of the polynomial $X^{p^d} - X$ in $\mathbb{F}_p[X]$. Conversely, every irreducible factor of $X^{p^d} - X$ which is not a factor of $X^{p^e} - X$ for $e < d$ has degree exactly d. This leads to the following algorithm.

Algorithm 3.4.3 (Distinct Degree Factorization). Given a squarefree polynomial $A \in \mathbb{F}_p[X]$, this algorithm finds for each d the polynomial A_d which is the product of the irreducible factors of A of degree d.

1. [Initialize] Set $V \leftarrow A$, $W \leftarrow X$, $d \leftarrow 0$.
2. [Finished?] Set $e \leftarrow \deg(V)$. If $d+1 > \frac{1}{2}e$, then if $e > 0$ set $A_e = V$, $A_i = 1$ for all other $i > d$, and terminate the algorithm. If $d+1 \leq \frac{1}{2}e$, set $d \leftarrow d+1$, $W \leftarrow W^p \bmod V$.
3. [Output A_d] Output $A_d = (W - X, V)$. If $A_d \neq 1$, set $V \leftarrow V/A_d$, $W \leftarrow W \bmod V$. Go to step 2.

Once the A_d have been found, it remains to factor them. We already know the number of irreducible factors of A_d, which is equal to $\deg(A_d)/d$. In particular, if $\deg(A_d) = d$, then A_d is irreducible.

Note that the distinct degree factorization algorithm above succeeds in factoring A completely quite frequently. With reasonable assumptions, it can

be shown that the irreducible factors of A modulo p will have distinct degrees with probability close to $e^{-\gamma} \approx 0.56146$, where γ is Euler's constant, where we assume the degree of A to be large (see [Knu2]).

As a corollary to the above discussion and algorithm, we see that it is easy to determine whether a polynomial is irreducible in $\mathbb{F}_p[X]$. More precisely, we have:

Proposition 3.4.4. *A polynomial $A \in \mathbb{F}_p[X]$ of degree n is irreducible if and only if the following two conditions are satisfied:*

$$X^{p^n} \equiv X \pmod{A(X)},$$

and for each prime q dividing n

$$(X^{p^{n/q}} - X, A(X)) = 1.$$

Note that to test in practice the second condition of the proposition, one must first compute $B(X) = X^{p^{n/q}} \bmod A(X)$ using the powering algorithm, and then compute $\gcd(B(X) - X, A(X))$. Hence, the time necessary for this irreducibility test, assuming one uses the $O(n^2)$ algorithms for multiplication and division of polynomials of degree n, is essentially $O(n^3 \ln p)$, if the factorization of n is known (since nobody considers polynomials of degree larger, say than 10^9, this is a reasonable assumption).

It is interesting to compare this with the analogous primality test for integers. By Proposition 8.3.1, n is prime if and only if for each prime q dividing $n-1$ one can find an $a_q \in \mathbb{Z}$ such that $a_q^{n-1} \equiv 1 \pmod{n}$ and $a_q^{(n-1)/q} \not\equiv 1 \pmod{n}$. This takes time $O(\ln^3 n)$, assuming the factorization of $n-1$ to be known. But this is an unreasonable assumption, since one commonly wants to prove the primality of numbers of 100 decimal digits, and at present it is quite unreasonable to factor a 100 digit number. Hence the above criterion is not useful as a general purpose primality test over the integers.

3.4.4 Final Splitting

Finally we must consider the most important and central part of Algorithm 3.4.1, its step 3, which in fact does most of the work. After the preceding steps we are left with the following problem. Given a polynomial A which is known to be squarefree and equal to a product of irreducible polynomials of degree exactly equal to d, find these factors. Of course, $\deg(A)$ is a multiple of d, and if $\deg(A) = d$ we know that A is itself irreducible and there is nothing to do. A simple and efficient way to do this was found by Cantor and Zassenhaus. Assume first that $p > 2$. Then we have the following lemma:

Proposition 3.4.5. *If A is as above, then for any polynomial $T \in \mathbb{F}_p[X]$ we have the identity:*

$$A = (A, T) \cdot (A, T^{(p^d-1)/2} + 1) \cdot (A, T^{(p^d-1)/2} - 1).$$

Proof. The roots of the polynomial $X^{p^d} - X$, being the elements of $\mathbb{F}_{p^d}$, are all distinct. It follows that for any $T \in \mathbb{F}_p[X]$, the polynomial $T(X)^{p^d} - T(X)$ also has all the elements of $\mathbb{F}_{p^d}$ as roots, hence is divisible by $X^{p^d} - X$. In particular, as we have seen in the preceding section, it is a multiple of every irreducible polynomial of degree d, hence of A, since A is squarefree. The claimed identity follows immediately by noting that

$$T^{p^d} - T = T \cdot (T^{(p^d-1)/2} + 1) \cdot (T^{(p^d-1)/2} - 1)$$

with the three factors pairwise coprime. □

Now it is not difficult to show that if one takes for T a random monic polynomial of degree less than or equal to $2d-1$, then $(A, T^{(p^d-1)/2} - 1)$ will be a non-trivial factor of A with probability close to $1/2$. Hence, we can use the following algorithm to factor A:

Algorithm 3.4.6 (Cantor-Zassenhaus Split). Given A as above, this algorithm outputs its irreducible factors (which are all of degree d). This algorithm will be called recursively.

1. [Initialize] Set $k \leftarrow \deg(A)/d$. If $k = 1$, output A and terminate the algorithm.
2. [Try a T] Choose $T \in \mathbb{F}_p[X]$ randomly such that T is monic of degree less than or equal to $2d - 1$. Set $B \leftarrow (A, T^{(p^d-1)/2} - 1)$. If $\deg(B) = 0$ or $\deg(B) = \deg(A)$ go to step 2.
3. [Recurse] Factor B and A/B using the present algorithm recursively, and terminate the algorithm.

Note that, as has already been mentioned after Proposition 3.4.4, to compute B in step 2 one first computes $C \leftarrow T^{(p^d-1)/2} \bmod A$ using the powering algorithm, and then $B \leftarrow (A, C - 1)$.

Finally, we must consider the case where $p = 2$. In that case, the following result is the analog of Proposition 3.4.5:

Proposition 3.4.7. *Set*

$$U(X) = X + X^2 + X^4 + \cdots + X^{2^{d-1}}.$$

If $p = 2$ and A is as above, then for any polynomial $T \in \mathbb{F}_2[X]$ we have the identity

$$A = (A, U \circ T) \cdot (A, U \circ T + 1).$$

Proof. Note that $(U \circ T)^2 = T^2 + T^4 + \cdots + T^{2^d}$, hence $(U \circ T) \cdot (U \circ T + 1) = T^{2^d} - T$ (remember that we are in characteristic 2). By the proof of Proposition 3.4.5 we know that this is a multiple of A, and the identity follows. □

Exactly as in the case of $p > 2$, one can show that the probability that $(A, U \circ T)$ is a non-trivial factor of A is close to $1/2$, hence essentially the same algorithm as Algorithm 3.4.6 can be used. Simply replace in step 2 $B \leftarrow (A, T^{(p^d-1)/2} - 1)$ by $B \leftarrow (A, U \circ T)$. Here, however, we can do better than choosing random polynomials T in step 2 as follows.

Algorithm 3.4.8 (Split for $p = 2$). Given $A \in \mathbb{F}_2[X]$ as above, this algorithm outputs its irreducible factors (which are all of degree d). This algorithm will be called recursively.

1. [Initialize] Set $k \leftarrow \deg(A)/d$. If $k = 1$, output A and terminate the algorithm, otherwise set $T \leftarrow X$.
2. [Test T] Set $C \leftarrow T$ and then repeat $d - 1$ times $C \leftarrow T + C^2 \bmod A$. Then set $B \leftarrow (A, C)$. If $\deg(B) = 0$ or $\deg(B) = \deg(A)$ then set $T \leftarrow T \cdot X^2$ and go to step 2.
3. [Recurse] Factor B and A/B using the present algorithm recursively, and terminate the algorithm.

Proof. If this algorithm terminates, it is clear that the output is a factorization of A, hence the algorithm is correct. We must show that it terminates. Notice first that the computation of C done in step 2 is nothing but the computation of $U \circ T \bmod A$ (note that on page 630 of [Knu2], Knuth gives $C \leftarrow (C + C^2 \bmod A)$, but this should be instead, as above, $C \leftarrow T + C^2 \bmod A$).

Now, since for any $T \in \mathbb{F}_2[X]$, we have by Proposition 3.4.7 $U(T) \cdot (U(T) + 1) \equiv 0 \pmod A$, it is clear that $(U(T), A) = 1$ is equivalent to $U(T) \equiv 1 \pmod A$. Furthermore, one immediately checks that $U(T^2) \equiv U(T) \pmod A$, and that $U(T_1 + T_2) = U(T_1) + U(T_2)$.

Now I claim that the algorithm terminates when $T = X^e$ in step 2 for some odd value of e such that $e \leq 2d - 1$. Indeed, assume the contrary. Then we have for every odd $e \leq 2d - 1$, $(U(X^e), A) = 1$ or A, hence $U(X^e) \equiv 0$ or 1 modulo A. Since $U(T^2) \equiv U(T) \pmod A$, this is true also for even values of $e \leq 2d$, and the linearity of U implies that this is true for every polynomial of degree less than or equal to $2d$. Now U is a polynomial of degree 2^{d-1}, and has at most (in fact exactly, see Exercise 15) 2^{d-1} roots in $\mathbb{F}_{2^d}$. Let $\beta \in \mathbb{F}_{2^d}$ not a root of U. The number of irreducible factors of A is at least equal to 2 (otherwise we would have stopped at step 1), and let A_1 and A_2 be two such factors, both of degree d. Let α be a root of A_2 in $\mathbb{F}_{2^d}$ (notice that all the roots of A_2 are in $\mathbb{F}_{2^d}$). Since A_2 is irreducible, α generates $\mathbb{F}_{2^d}$ over $\mathbb{F}_2$. Hence, there exists a polynomial $P \in \mathbb{F}_2[X]$ such that $\beta = P(\alpha)$.

By the Chinese remainder theorem, since A_1 and A_2 are coprime we can choose a polynomial T such that $T \equiv 0 \pmod{A_1}$ and $T \equiv P \pmod{A_2}$, and

T is defined modulo the product A_1A_2. Hence, we can choose T of degree less than $2d$. But

$$U(T) \equiv U(0) \equiv 0 \pmod{A_1}$$

and

$$U(T) \equiv U(P) \not\equiv 0 \pmod{A_2}$$

since

$$U(P(\alpha)) = U(\beta) \neq 0.$$

This contradicts $U(T) \equiv 0$ or 1 modulo A, thus proving the validity of the algorithm. The same proof applied to $T^{p^d} - T$ instead of $U(T)$ explains why one can limit ourselves to $\deg(T) \leq 2d-1$ in Algorithm 3.4.6. □

Proposition 3.4.7 and Algorithm 3.4.8 can be extended to general primes p, but are useful in practice only if p is small (see Exercise 14).

There is another method for doing the final splitting due to Berlekamp which predates that of Cantor-Zassenhaus, and which is better in many cases. This method could be used as soon as the polynomial is squarefree. (In other words, if desirable, we can skip the distinct degree factorization.) It is based on the following proposition.

Proposition 3.4.9. *Let $A \in \mathbb{F}_p[X]$ be a squarefree polynomial, and let*

$$A(X) = \prod_{1 \leq i \leq r} A_i(X)$$

be its decomposition into irreducible factors. The polynomials $T \in \mathbb{F}_p[X]$ with $\deg(T) < \deg(A)$ for which for each i with $1 \leq i \leq r$ there exist $s_i \in \mathbb{F}_p$ such that $T(X) \equiv s_i \pmod{A_i(X)}$, are exactly the p^r polynomials T such that $\deg(T) < \deg(A)$ and $T(X)^p \equiv T(X) \pmod{A(X)}$.

Proof. By the Chinese remainder Theorem 1.3.9 generalized to the Euclidean ring $\mathbb{F}_p[X]$, for each of the p^r possible choices of $s_i \in F_p$ $(1 \leq i \leq r)$, there exists a unique polynomial $T \in \mathbb{F}_p[X]$ such that $\deg(T) < \deg(A)$ and for each i

$$T(X) \equiv s_i \pmod{A_i(X)}.$$

Now if T is a solution of such a system, we have

$$T(X)^p \equiv s_i^p = s_i \equiv T(X) \pmod{A_i(X)}$$

for each i, hence

$$T(X)^p \equiv T(X) \pmod{A(X)}.$$

Conversely, note that we have in $\mathbb{F}_p[X]$ the polynomial identity $X^p - X = \prod_{0 \leq s \leq p-1}(X - s)$, hence

$$T(X)^p - T(X) = \prod_{0 \le s \le p-1} (T(X) - s).$$

Hence, if $T(X)^p \equiv T(X) \pmod{A(X)}$, we have for all i

$$A_i(X) \mid \prod_{0 \le s \le p-1} (T(X) - s),$$

and since the A_i are irreducible this means that $A_i(X) \mid T(X) - s_i$ for some $s_i \in \mathbb{F}_p$ thus proving the proposition. □

The relevance of this proposition to our splitting problem is the following. If T is a solution of such a system of congruences with, say, $s_1 \neq s_2$, then $\gcd(A(X), T(X) - s_1)$ will be divisible by A_1 and not by A_2, hence it will be a non-trivial divisor of A. The above proposition tells us that to look for such nice polynomials T it is not necessary to know the A_i, but simply to solve the congruence $T(X)^p \equiv T(X) \pmod{A(X)}$.

To solve this, write $T(X) = \sum_{0 \le j < n} t_j X^j$, where $n = \deg(A)$, with $t_j \in \mathbb{F}_p$. Then $T(X)^p = \sum_j t_j X^{pj}$, hence if we set

$$X^{pk} \equiv \sum_{0 \le i < n} q_{i,k} X^i \pmod{A(X)}$$

we have

$$T(X)^p \equiv \sum_j t_j \sum_i q_{i,j} X^i \pmod{A(X)}$$

so the congruence $T(X)^p \equiv T(X) \pmod{A(X)}$ is equivalent to

$$\sum_j t_j q_{i,j} = t_i \quad \text{for} \quad 1 \le i < n.$$

If, in matrix terms, we set $Q = (q_{i,j})$, $V = (t_j)$ (column vector), and I the identity matrix, this means that $QV = V$. In other words $(Q - I)V = 0$, so V belongs to the kernel of the matrix $Q - I$.

Algorithm 2.3.1 will allow us to compute this kernel, and it is especially efficient since we work in a finite field where no coefficient explosion or instability occurs.

Thus we will obtain a basis of the kernel of $Q - I$, which will be of dimension r by Proposition 3.4.9. Note that trivially $q_{i,0} = 0$ if $i > 0$ and $q_{0,0} = 1$, hence the column vector $(1, 0, \ldots, 0)^t$ will always be an element of the kernel, corresponding to the trivial choice $T(X) = 1$. Any other basis element of the kernel will be useful. If $T(X)$ is the polynomial corresponding to a V in the kernel of $Q - I$, we compute $(A(X), T(X) - s)$ for $0 \le s \le p - 1$. Since by Proposition 3.4.9 there exists an s such that $T(X) \equiv s \pmod{A_1(X)}$, there will exist an s which will give a non-trivial GCD, hence a splitting of A. We

apply this to all values of s and all basis vectors of the kernel until the r irreducible factors of A have been isolated (note that it is better to proceed in this way than to use the algorithm recursively once a split is found as in Algorithm 3.4.6 since it avoids the recomputation of Q and of the kernel of $Q - I$).

This leads to the following algorithm.

Algorithm 3.4.10 (Berlekamp for Small p). Given a squarefree polynomial $A \in \mathbb{F}_p[X]$ of degree n, this algorithm computes the factorization of A into irreducible factors.

1. [Compute Q] Compute inductively for $0 \leq k < n$ the elements $q_{i,k} \in \mathbb{F}_p$ such that
$$X^{pk} \equiv \sum_{0 \leq i < n} q_{i,k} X^i \pmod{A(X)}.$$
2. [Compute kernel] Using Algorithm 2.3.1, find a basis $V_1, \ldots, V_r$ of the kernel of $Q - I$. Then r will be the number of irreducible factors of A, and $V_1 = (1, 0, \ldots, 0)^t$. Set $E \leftarrow \{A\}$, $k \leftarrow 1$, $j \leftarrow 1$ (E will be a set of polynomials whose product is equal to A, k its cardinality and j is the index of the vector of the kernel which we will use).
3. [Finished?] If $k = r$, output E as the set of irreducible factors of A and terminate the algorithm. Otherwise, set $j \leftarrow j + 1$, and let $T(X)$ be the polynomial corresponding to the vector V_j (i.e. $T(X) \leftarrow \sum_{0 \leq i < n} (V_j)_i X^i$).
4. [Split] For each element $B \in E$ such that $\deg(B) > 1$ do the following. For each $s \in \mathbb{F}_p$ compute $(B(X), T(X) - s)$. Let F be the set of such GCD's whose degree is greater or equal to 1. Set $E \leftarrow (E \setminus \{B\}) \cup F$ and $k \leftarrow k - 1 + |F|$. If in the course of this computation we reach $k = r$, output E and terminate the algorithm. Otherwise, go to step 3.

The main drawback of this algorithm is that the running time of step 4 is proportional to p, and this is slower than Algorithm 3.4.6 as soon as p gets above 100 say. On the other hand, if p is small, a careful implementation of this algorithm will be faster than Algorithm 3.4.6. This is important, since in many applications such as factoring polynomials over $\mathbb{Z}$, we will first factor the polynomial over a few fields $\mathbb{F}_p$ for small primes p where Berlekamp's algorithm is superior.

If we use the idea of the Cantor-Zassenhaus split, we can however improve considerably Berlekamp's algorithm when p is large. In steps 3 and 4, instead of considering the polynomials corresponding to the vectors $V_j - sV_1$ for $2 \leq j \leq r$ and $s \in \mathbb{F}_p$, we instead choose a random linear combination $V = \sum_{i=1}^{r} a_i V_i$ with $a_i \in \mathbb{F}_p$ and compute $(B(X), T(X)^{(p-1)/2} - 1)$, where T is the polynomial corresponding to V. It is easy to show that this GCD will give a non-trivial factor of $B(X)$ with probability greater than or equal to $4/9$ when $p \geq 3$ and B is reducible (see Exercise 17 and [Knu2] p. 429). This gives the following algorithm.

Algorithm 3.4.11 (Berlekamp). Given a squarefree polynomial $A \in \mathbb{F}_p[X]$ of degree n (with $p \geq 3$), this algorithm computes the factorization of A into irreducible factors.

1. [Compute Q] Compute inductively for $0 \leq k < n$ the elements $q_{i,k} \in \mathbb{F}_p$ such that
$$X^{pk} \equiv \sum_{0 \leq i < n} q_{i,k} X^i.$$
2. [Compute kernel] Using Algorithm 2.3.1, find a basis $V_1, \ldots, V_r$ of the kernel of $Q - I$, and let $T_1, \ldots, T_r$ be the corresponding polynomials. Then r will be the number of irreducible factors of A, and $V_1 = (1, 0, \ldots, 0)^t$ hence $T_1 = 1$. Set $E \leftarrow \{A\}$, $k \leftarrow 1$, (E will be a set of polynomials whose product is equal to A and k its cardinality).
3. [Finished?] If $k = r$, output E as the set of irreducible factors of A and terminate the algorithm. Otherwise, choose r random elements $a_i \in \mathbb{F}_p$, and set $T(X) \leftarrow \sum_{1 \leq i \leq r} a_i T_i(X)$.
4. [Split] For each element $B \in E$ such that $\deg(B) > 1$ do the following. Let $D(X) \leftarrow (B(X), T(X)^{(p-1)/2} - 1)$. If $\deg(D) > 0$ and $\deg(D) < \deg(B)$, set $E \leftarrow (E \backslash \{B\}) \cup \{D, B/D\}$ and $k \leftarrow k+1$. If in the course of this computation we reach $k = r$, output E and terminate the algorithm. Otherwise, go to step 3.

Note that if we precede any of these two Berlekamp algorithms by the distinct degree factorization procedure (Algorithm 3.4.3), we should replace the condition $\deg(B) > 1$ of step 4 by $\deg(B) > d$, since we know that all the irreducible factors of A have degree d.

Using the algorithms of this section, we now have at our disposal several efficient methods for completely factoring polynomials modulo a prime p. We will now consider the more difficult problem of factoring over $\mathbb{Z}$.

3.5 Factorization of Polynomials over $\mathbb{Z}$ or $\mathbb{Q}$

The first thing to note is that factoring over $\mathbb{Q}$ is essentially equivalent to factoring over $\mathbb{Z}$. Indeed if $A = \prod_i A_i$ where the A_i are irreducible over $\mathbb{Q}$, then by multiplying by suitable rational numbers, we have $dA = \prod_i (d_i A_i)$ where the d_i can be chosen so that the $d_i A_i$ have integer coefficients and are primitive. Hence it follows from Gauss's lemma (Theorem 3.2.8) that if $A \in \mathbb{Z}[X]$, then $d = \pm 1$. Conversely, if $A = \prod_i A_i$ with A and the A_i in $\mathbb{Z}[X]$ and the A_i are irreducible over $\mathbb{Z}$, then the A_i are also irreducible over $\mathbb{Q}$, by a similar use of Gauss's lemma.

Therefore in this section, we will consider only the problem of factoring a polynomial A over $\mathbb{Z}$. If $A = BC$ is a splitting of A in $\mathbb{Z}[X]$, then $\bar{A} = \bar{B}\bar{C}$ in $\mathbb{F}_p[X]$, where $\bar{}$ denotes reduction mod p. Hence we can start by reducing mod p for some p, factor mod p, and then see if the factorization over $\mathbb{F}_p$ lifts to one over $\mathbb{Z}$. For this, it is essential to know an upper bound on the absolute value of the coefficients which can occur as a factor of A.

3.5.1 Bounds on Polynomial Factors

The results presented here are mostly due to Mignotte [Mig]. The aim of this section is to prove the following theorem:

Theorem 3.5.1. *For any polynomial* $P = \sum_{0\le i\le n} p_i X^i \in \mathbb{C}[X]$ *set* $|P| = (\sum_i |p_i|^2)^{1/2}$. *Let* $A = \sum_{0\le i\le m} a_i X^i$ *and* $B = \sum_{0\le i\le n} b_i X^i$ *be polynomials with integer coefficients, and assume that* B *divides* A. *Then we have for all* j

$$|b_j| \le \binom{n-1}{j}|A| + \binom{n-1}{j-1}|a_m|.$$

Proof. Let α be any complex number, and let $A = \sum_{0\le i\le m} a_i X^i$ be any polynomial. Set $G(X) = (X-\alpha)A(X)$ and $H(X) = (\overline{\alpha}X - 1)A(X)$. Then

$$\begin{aligned}|G|^2 &= \sum |a_{i-1} - \alpha a_i|^2 = \sum (|a_{i-1}|^2 + |\alpha a_i|^2 - 2\,\mathrm{Re}(\alpha a_i \overline{a_{i-1}})) \\ &= \sum (|\alpha a_{i-1}|^2 + |a_i|^2 - 2\,\mathrm{Re}(\alpha a_i \overline{a_{i-1}})) \\ &= \sum |\overline{\alpha} a_{i-1} - a_i|^2 = |H|^2.\end{aligned}$$

Let now $A(X) = a_m \prod_j (X - \alpha_j)$ be the decomposition of A over $\mathbb{C}$. If we set

$$C(X) = a_m \prod_{|\alpha_j|\ge 1} (X - \alpha_j) \prod_{|\alpha_j|<1} (\overline{\alpha_j}X - 1),$$

it follows that $|C| = |A|$. Hence, taking into account only the coefficient of X^m and the constant term, it follows that

$$|A|^2 = |C|^2 \ge |a_m|^2 (M(A)^2 + m(A)^2),$$

where

$$M(A) = \prod_{|\alpha_j|>1} |\alpha_j|, \qquad m(A) = \prod_{|\alpha_j|<1} |\alpha_j|.$$

In particular, $M(A) \le |A|/|a_m|$. Now

$$|a_j| = |a_m| \left| \sum \alpha_{i_1} \dots \alpha_{i_{m-j}} \right| \le |a_m| \sum \beta_{i_1} \dots \beta_{i_{m-j}},$$

where $\beta_i = \max(1, |\alpha_i|)$. Assume for the moment the following lemma:

Lemma 3.5.2. *If* $x_1 \ge 1, \dots, x_m \ge 1$ *are real numbers constrained by the further condition that their product is equal to* M, *then the elementary symmetric function* $\sigma_{mk} = \sum x_{i_1} \dots x_{i_k}$ *satisfies*

$$\sigma_{mk} \le \binom{m-1}{k-1} M + \binom{m-1}{k}.$$

Since the product of the β_i is by definition $M(A)$, it follows from the lemma that for all j,

$$|a_j| \leq |a_m| \left(\binom{m-1}{m-j-1} M(A) + \binom{m-1}{m-j} \right)$$
$$\leq |a_m| \left(\binom{m-1}{j} M(A) + \binom{m-1}{j-1} \right).$$

Coming back to our notations and applying the preceding result to the polynomial B, we see that $|b_j| \leq |b_n|(\binom{n-1}{j} M(B) + \binom{n-1}{j-1})$. It follows that $|b_j| \leq |a_m|(\binom{n-1}{j} M(A) + \binom{n-1}{j-1})$ since if B divides A, we must have $M(B) \leq M(A)$ (since the roots of B are roots of A), and $|b_n| \leq |a_m|$ (since in fact b_n divides a_m). The theorem follows from this and the inequality $M(A) \leq |A|/|a_m|$ proved above.

It remains to prove the lemma. Assume without loss of generality that $x_1 \leq x_2 \cdots \leq x_m$. If one changes the pair (x_{m-1}, x_m) into the pair $(1, x_{m-1}x_m)$, all the constraints are still satisfied and it is easy to check that the value of σ_{mk} is increased by

$$\sigma_{(m-2)(k-1)}(x_{m-1} - 1)(x_m - 1).$$

It follows that if $x_{m-1} > 1$, the point $(x_1, \ldots, x_m)$ cannot be a maximum. Hence a necessary condition for a maximum is that $x_{m-1} = 1$. But this immediately implies that $x_i = 1$ for all $i < m$, and hence that $x_m = M$. It is now a simple matter to check the inequality of the lemma, the term $\binom{m-1}{k-1} M$ corresponding to k-tuples containing x_m, and the term $\binom{m-1}{k}$ to the ones which do not contain x_m. This finishes the proof of Theorem 3.5.1. □

A number of improvements can be made in the estimates given by this theorem. They do not seriously influence the running time of the algorithms using them however, hence we will be content with this.

3.5.2 A First Approach to Factoring over $\mathbb{Z}$

First note that for polynomials A of degree 2 or 3 with coefficients which are not too large, the factoring problem is easy: if A is not irreducible, it must have a linear factor $qX - p$, and q must divide the leading term of A, and p must divide the constant term. Hence, if the leading term and the constant term can be easily factored, one can check each possible divisor of A. An ad hoc method of this sort could be worked out also in higher degrees, but soon becomes impractical.

A second way of factoring over $\mathbb{Z}$ is to combine information obtained by the mod p factorization methods. For example, if modulo some prime p, $A(X) \bmod p$ is irreducible, then $A(X)$ itself is of course irreducible. A less trivial example is the following: if for some p a polynomial $A(X)$ of degree 4 breaks modulo p into a product of two irreducible polynomials of degree 2,

and for another p into a product of a polynomial of degree 1 and an irreducible polynomial of degree 3, then $A(X)$ must be irreducible since these splittings are incompatible. Unfortunately, although this method is useful when combined with other methods, except for polynomials of small degree, when used alone it rarely works. For example, using the quadratic reciprocity law and the identities

$$\begin{aligned} X^4+1 &= (X^2+\sqrt{-1})(X^2-\sqrt{-1}) \\ &= (X^2-X\sqrt{2}+1)(X^2+X\sqrt{2}+1) \\ &= (X^2+X\sqrt{-2}-1)(X^2-X\sqrt{-2}-1) \end{aligned}$$

it is easy to check that the polynomial X^4+1 splits into 4 linear factors if $p=2$ or $p \equiv 1 \pmod 8$, and into two irreducible quadratic factors otherwise. This is compatible with the possibility that X^4+1 could split into 2 quadratic factors in $\mathbb{Z}[X]$, and this is clearly not the case.

A third way to derive a factorization algorithm over $\mathbb{Z}$ is to use the bounds given by Theorem 3.5.1 and the mod p factorization methods. Consider for example the polynomial

$$A(X) = X^6 - 6X^4 - 2X^3 - 7X^2 + 6X + 1.$$

If it is not irreducible, it must have a factor of degree at most 3. The bound of Theorem 3.5.1 shows that for any factor of degree less or equal to 3 and any j, one must have $|b_j| \leq 23$. Take now a prime p greater than twice that bound and for which the polynomial A mod p is squarefree, for example $p=47$. The mod p factoring algorithms of the preceding section show that modulo 47 we have

$$A(X) = (X-22)(X-13)(X-12)(X+12)(X^2-12X-4),$$

taking as representatives of $\mathbb{Z}/47\mathbb{Z}$ the numbers from -23 to 23. Now the constant term of A being equal to 1, up to sign any factor of A must have that property. This immediately shows that A has no factor of degree 1 over $\mathbb{Z}$ (this could of course have been checked more easily simply by noticing that $A(1)$ and $A(-1)$ are both non-zero), but it also shows that A has no factor of degree 2 since modulo 47 we have $12 \cdot 22 = -18$, $12 \cdot 13 = 15$, $12 \cdot 12 = 3$ and $13 \cdot 22 = 4$. Hence, if A is reducible, the only possibility is that A is a product of two factors of degree 3. One of them must be divisible by $X^2 - 12X - 4$, and hence can be (modulo 47) equal to either $(X^2-12X-4)(X-12)$ (whose constant term is 1), or to $(X^2-12X-4)(X+12)$ (whose constant term is -1). Now modulo 47, we have $(X^2-12X-4)(X-12) = X^3+23X^2-X+1$ and $(X^2-12X-4)(X+12) = X^3-7X-1$.

The first case can be excluded a priori because the bound of Theorem 3.5.1 gives $b_2 \leq 12$, hence 23 is too large. In the other case, by the choice made for p, this is the only polynomial in its congruence class modulo 47 satisfying the bounds of Theorem 3.5.1. Hence, if it divides A in $\mathbb{Z}[X]$, we have found the

factorization of A, otherwise we can conclude that A is irreducible. Since one checks that $A(X) = (X^3 - 7X - 1)(X^3 + X - 1)$, we have thus obtained the complete factorization of A over $\mathbb{Z}$. Note that the irreducibility of the factors of degree 3 has been proved along the way.

When the degree or the coefficients of A are large however, the bounds of Theorem 3.5.1 imply that we must use a p which is really large, and hence for which the factorization modulo p is too slow. We can overcome this problem by keeping a small p, but factoring modulo p^e for sufficiently large e.

3.5.3 Factorization Modulo p^e: Hensel's Lemma

The trick is that if certain conditions are satisfied, we can "lift" a factorization modulo p to a factorization mod p^e for any desired e, without too much effort. This is based on the following theorem, due to Hensel, and which was one of his motivations for introducing p-adic numbers.

Theorem 3.5.3. *Let p be a prime, and let C, A_e, B_e, U, V be polynomials with integer coefficients and satisfying*

$$C(X) \equiv A_e(X)B_e(X) \pmod{p^e}, \quad U(X)A_e(X)+V(X)B_e(X) \equiv 1 \pmod{p}.$$

Assume that $e \geq 1$, A_e is monic, $\deg(U) < \deg(B_e)$, $\deg(V) < \deg(A_e)$. Then there exist polynomials A_{e+1} and B_{e+1} satisfying the same conditions with e replaced by $e+1$, and such that

$$A_{e+1}(X) \equiv A_e(X) \pmod{p^e}, \qquad B_{e+1}(X) \equiv B_e(X) \pmod{p^e}.$$

Furthermore, these polynomials are unique modulo p^{e+1}.

Proof. Set $D = (C - A_eB_e)/p^e$ which has integral coefficients by assumption. We must have $A_{e+1} = A_e + p^eS$, $B_{e+1} = B_e + p^eT$ with S and T in $\mathbb{Z}[X]$. The main condition needed is $C(X) \equiv A_{e+1}(X)B_{e+1}(X) \pmod{p^{e+1}}$. Since $2e \geq e+1$, this is equivalent to $A_eT + B_eS \equiv (C - A_eB_e)/p^e = D \pmod{p}$. Since $UA_e + VB_e = 1$ in $\mathbb{F}_p[X]$ and $\mathbb{F}_p$ is a field, the general solution is $S \equiv VD + WA_e \pmod{p}$ and $T \equiv UD - WB_e) \pmod{p}$ for some polynomial W. The conditions on the degrees imply that S and T are unique modulo p, hence A_{e+1} and B_{e+1} are unique modulo p^{e+1}. Note that this proof is constructive, and gives a simple algorithm to obtain A_{e+1} and B_{e+1}. □

For reasons of efficiency, it will be useful to have a more general version of Theorem 3.5.3. The proof is essentially identical to the proof of Theorem 3.5.3, and will follow from the corresponding algorithm.

Theorem 3.5.4. *Let p, q be (not necessarily prime) integers, and let $r = (p, q)$. Let C, A, B, U and V be polynomials with integer coefficients satisfying*

$$C \equiv AB \pmod{q}, \qquad UA + VB \equiv 1 \pmod{p},$$

and assume that $(\ell(A), r) = 1$, $\deg(U) < \deg(B)$, $\deg(V) < \deg(A)$ *and* $\deg(C) = \deg(A) + \deg(B)$. *(Note that this last condition is not automatically satisfied since* $\mathbb{Z}/q\mathbb{Z}$ *may have zero divisors.) Then there exist polynomials* A_1 *and* B_1 *such that* $A_1 \equiv A \pmod{q}$, $B_1 \equiv B \pmod{q}$, $\ell(A_1) = \ell(A)$, $\deg(A_1) = \deg(A)$, $\deg(B_1) = \deg(B)$ *and*

$$C \equiv A_1 B_1 \pmod{qr}.$$

In addition, A_1 *and* B_1 *are unique modulo* qr *if* r *is prime.*

We give the proof as an algorithm.

Algorithm 3.5.5 (Hensel Lift). Given the assumptions and notations of Theorem 3.5.4, this algorithm outputs the polynomials A_1 and B_1. As a matter of notation, we denote by K the ring $\mathbb{Z}/r\mathbb{Z}$.

1. [Euclidean division] Set $f \leftarrow (C - AB)/q \pmod{r} \in K[X]$. Using Algorithm 3.1.1 over the ring K, find $t \in K[X]$ such that $\deg(Vf - At) < \deg(A)$ (this is possible even when K is not a field, since $\ell(A)$ is invertible in K).
2. [Terminate] Let A_0 be a lift of $Vf - At$ to $\mathbb{Z}[X]$ having the same degree, and let B_0 be a lift of $Uf + Bt$ to $\mathbb{Z}[X]$ having the same degree. Output $A_1 \leftarrow A + qA_0$, $B_1 \leftarrow B + qB_0$ and terminate the algorithm.

Proof. It is clear that $BA_0 + AB_0 \equiv f \pmod{r}$. From this, it follows immediately that $C \equiv A_1 B_1 \pmod{qr}$ and that $\deg(B_0) \leq \deg(B)$, thus proving the validity of the algorithm and of Theorem 3.5.4. □

If $p \mid q$, we can also if desired replace p by $pr = p^2$ in the following way.

Algorithm 3.5.6 (Quadratic Hensel Lift). Assume $p \mid q$, hence $r = p$. After execution of Algorithm 3.5.5, this algorithm finds U_1 and V_1 such that $U_1 \equiv U \pmod{p}$, $V_1 \equiv V \pmod{p}$, $\deg(U_1) < \deg(B_1)$, $\deg(V_1) < \deg(A_1)$ and

$$U_1 A_1 + V_1 B_1 \equiv 1 \pmod{pr}.$$

1. [Euclidean division] Set $g \leftarrow (1 - UA_1 - VB_1)/p \pmod{r}$. Using Algorithm 3.1.1 over the same ring $K = \mathbb{Z}/r\mathbb{Z}$, find $t \in K[X]$ such that $\deg(Vg - A_1 t) < \deg(A_1)$, which is possible since $\ell(A_1) = \ell(A)$ is invertible in K.
2. [Terminate] Let U_0 be a lift of $Ug + B_1 t$ to $\mathbb{Z}[X]$ having the same degree, and let V_0 be a lift of $Vg - A_1 t$ to $\mathbb{Z}[X]$ having the same degree. Output $U_1 \leftarrow U + pU_0$, $V_1 \leftarrow V + pV_0$ and terminate the algorithm.

It is not difficult to see that at the end of this algorithm, (A_1, B_1, U_1, V_1) satisfy the same hypotheses as (A, B, U, V) in the theorem, with (p, q) replaced by (pr, qr).

The condition $p \mid q$ is necessary for Algorithm 3.5.6 (not for Algorithm 3.5.5), and was forgotten by Knuth (page 628). Indeed, if $p \nmid q$, G does not have integral coefficients in general, since after constructing A_1 and B_1, one has only the congruence $UA_1+VB_1 \equiv 1 \pmod{r}$ and not $\pmod{p}$. Of course, this only shows that Algorithm 3.5.6 cannot be used in that case, but it does not show that it is impossible to find U_1 and V_1 by some other method. It is however easy to construct counterexamples. Take $p = 33$, $q = 9$, hence $r = 3$, and $A(X) = X - 3$, $B(X) = X - 4$, $C(X) = X^2 + 2X + 3$, $U(X) = 1$ and $V(X) = -1$. The conditions of the theorem are satisfied, and Algorithm 3.5.5 gives us $A_1(X) = X - 21$ and $B_1(X) = X + 23$. Consider now the congruence that we want, i.e.

$$U_1(X)(X - 21) + V_1(X)(X + 23) \equiv 1 \pmod{99},$$

or equivalently

$$U_1(X)(X - 21) + V_1(X)(X + 23) = 1 + 99W(X),$$

where all the polynomials involved have integral coefficients. If we set $X = 21$, we obtain $44V_1(21) = 1 + 99W(21)$, hence $0 \equiv 1 \pmod{11}$ which is absurd. This shows that even without any restriction on the degrees, it is not always possible to lift p to pr if $p \nmid q$.

The advantage of using both algorithms instead of one is that we can increase the value of the exponent e much faster. Assume that we start with $p = q$. Then, by using Algorithm 3.5.5 alone, we keep p fixed, and q takes the successive values p^2, p^3, etc .... If instead we use both Algorithms 3.5.5 and 3.5.6, the pair (p, q) takes the successive values (p^2, p^2), (p^4, p^4), etc ... with the exponent doubling each time. In principle this is much more efficient. When the exponent gets large however, the method slows down because of the appearance of multi-precision numbers. Hence, Knuth suggests the following recipe: let E be the smallest power of 2 such that p^E cannot be represented as a single precision number, and e be the largest integer such that p^e is a single precision number. He suggests working successively with the following pairs (p, q):

(p, p), (p^2, p^2), (p^4, p^4), ..., $(p^{E/2}, p^{E/2})$ using both algorithms, then (p^e, p^E) using both algorithms again but a reduced value of the exponent of p (since $e < E$) and finally (p^e, p^{E+e}), (p^e, p^{E+2e}), (p^e, p^{E+3e}), ... using only Algorithm 3.5.5.

Finally, note that by induction, one can extend Algorithms 3.5.5 and 3.5.6 to the case where C is congruent to a product of more than 2 pairwise coprime polynomials mod p.

3.5.4 Factorization of Polynomials over $\mathbb{Z}$

We now have enough ingredients to give a reasonably efficient method for factoring polynomials over the integers as follows.

Algorithm 3.5.7 (Factor in $\mathbb{Z}[X]$). Let $A \in \mathbb{Z}[X]$ be a non-zero polynomial. This algorithm finds the complete factorization of A in $\mathbb{Z}[X]$.

1. [Reduce to squarefree and primitive] Set $c \leftarrow \mathrm{cont}(A)$, $A \leftarrow A/c$, $U \leftarrow A/(A, A')$ where (A, A') is computed using the sub-resultant Algorithm 3.3.1, or the method of Section 3.6.1 below. (Now U will be a squarefree primitive polynomial. In this step, we could also use the squarefree decomposition Algorithm 3.4.2 to reduce still further the degree of U).

2. [Find a squarefree factorization mod p] For each prime p, compute (U, U') over the field $\mathbb{F}_p$, and stop when this GCD is equal to 1. For this p, using the algorithms of Section 3.4, find the complete factorization of U mod p (which will be squarefree). Note that in this squarefree factorization it is not necessary to find each A_j from the U_j: we will have $A_j = U_j$ since $T = (U, U') = 1$.

3. [Find bound] Using Theorem 3.5.1, find a bound B for the coefficients of factors of U of degree less than or equal to $\deg(U)/2$. Choose e to be the smallest exponent such that $p^e > 2\ell(U)B$.

4. [Lift factorization] Using generalizations of Algorithms 3.5.5 and 3.5.6, and the procedure explained in the preceding section, lift the factorization obtained in step 2 to a factorization mod p^e. (One will also have to use Euclid's extended Algorithm 3.2.2.) Let

$$U \equiv \ell(U)U_1U_2 \dots U_r \pmod{p^e}$$

   be the factorization of U mod p^e, where we can assume the U_i to be monic. Set $d \leftarrow 1$.

5. [Try combination] For every combination of factors $\bar{V} = U_{i_1} \dots U_{i_d}$, where in addition we take $i_d = 1$ if $d = \frac{1}{2}r$, compute the unique polynomial $V \in \mathbb{Z}[X]$ such that all the coefficients of V are in $[-\frac{1}{2}p^e, \frac{1}{2}p^e[$, and satisfying $V \equiv \ell(U)\bar{V} \pmod{p^e}$ if $\deg(V) \leq \frac{1}{2}\deg(U)$, $V \equiv U/\bar{V} \pmod{p^e}$ if $\deg(V) > \frac{1}{2}\deg(U)$.
   If V divides $\ell(U)U$ in $\mathbb{Z}[X]$, output the factor $F = \mathrm{pp}(V)$, the exponent of F in A, set $U \leftarrow U/F$, and remove the corresponding U_i from the list of factors mod p^e (i.e. remove $U_{i_1} \dots U_{i_d}$ and set $r \leftarrow r - d$ if $d \leq \frac{1}{2}r$, or leave only these factors and set $r \leftarrow d$ otherwise). If $d > \frac{1}{2}r$ terminate the algorithm by outputting $\mathrm{pp}(U)$ if $\deg(U) > 0$.

6. Set $d \leftarrow d + 1$. If $d \leq \frac{1}{2}r$ go to step 5, otherwise terminate the algorithm by outputting $\mathrm{pp}(U)$ if $\deg(U) > 0$.

Implementation Remarks. To decrease the necessary bound B, it is a good idea to reverse the coefficients of the polynomial U if $|u_0| < |u_n|$ (where of course we have cast out all powers of X so that $u_0 \neq 0$). Then the factors will be the reverse of the factors found.

In step 5, before trying to see whether V divides $\ell(U)U$, one should first test the divisibility of the constant terms, i.e. whether $V(0) \mid (\ell(U)U(0))$, since this will be rarely satisfied in general.

An important improvement can be obtained by using the information gained by factoring modulo a few small primes as mentioned in the second paragraph of Section 3.5.2. More precisely, apply the distinct degree factorization Algorithm 3.4.3 to U modulo a number of primes p_k (Musser and Knuth suggest about 5). If d_j are the degrees of the factors (it is not necessary to obtain the factors themselves) repeated with suitable multiplicity (so that $\sum_j d_j = n = \deg(U)$), build a binary string D_k of length $n+1$ which represents the degrees of all the possible factors mod p_k in the following way: Set $D_k \leftarrow (0 \ldots 01)$, representing the set with the unique element $\{0\}$. Then, for every d_j set

$$D_k \leftarrow D_k \vee (D_k \uparrow d_j) \ ,$$

where $\vee$ is inclusive "or", and $D_k \uparrow d_j$ is D_k shifted left d_j bits. (If desired, one can work with only the rightmost $\lceil (n+1)/2 \rceil$ bits of this string by symmetry of the degrees of the factors.)

Finally compute $D \leftarrow \bigwedge D_k$, i.e. the logical "and" of the bit strings. If the binary string D has only one bit at each end, corresponding to factors of degree 0 and n, this already shows that U is irreducible. Otherwise, choose for p the p_k giving the least number of factors. Then, during the execution of step 5 of Algorithm 3.5.7, keep only those d-uplets $(i_1, \ldots, i_d)$ such that the bit number $\deg(U_{i_1}) + \cdots + \deg(U_{i_d})$ of D is equal to 1.

Note that the prime chosen to make the Hensel lift will usually be small (say less than 20), hence in the modulo p factorization part, it will probably be faster to use Algorithm 3.4.10 than Algorithm 3.4.6 for the final splitting.

3.5.5 Discussion

As one can see, the problem of factoring over $\mathbb{Z}$ (or over $\mathbb{Q}$, which is essentially equivalent) is quite a difficult problem, and leads to an extremely complex algorithm, where there is a lot of room for improvement. Since this algorithm uses factorization mod p as a sub-algorithm, it is probabilistic in nature. Even worse, the time spent in step 5 above can be exponential in the degree. Therefore, a priori, the running time of the above algorithm is exponential in the degree. Luckily, in practice, its average behavior is random polynomial time. One should keep in mind however that in the worst case it *is* exponential time.

An important fact, discovered only relatively recently (1982) by Lenstra, Lenstra and Lovász is that it is possible to factor a polynomial over $\mathbb{Z}[X]$ in polynomial time using a deterministic algorithm. This is surprising in view of the corresponding problem over $\mathbb{Z}/p\mathbb{Z}[X]$ which should be simpler, and for which no such deterministic polynomial time algorithm is known, at least without assuming the Generalized Riemann Hypothesis. Their method uses in a fundamental way the LLL algorithm seen in Section 2.6.

The problem with the LLL factoring method is that, although in theory it is very nice, in practice it seems that it is quite a lot slower than the algorithm presented above. Therefore we will not give it here, but refer the

interested reader to [LLL]. Note also that A. K. Lenstra has shown that similar algorithms exist over number fields, and also for multivariate polynomials.

There is however a naïve way to apply LLL which gives reasonably good results. Let A be the polynomial to be factored, and assume as one may, that it is squarefree (but not necessarily primitive). Then compute the roots α_i of A in $\mathbb{C}$ with high accuracy (say 19 decimal digits) (for example using Algorithm 3.6.6 below), then apply Algorithm 2.7.4 to $1, \alpha, \ldots, \alpha^{k-1}$ for some $k < n$, where α is one of the α_i. Then if A is not irreducible, and if the constant N of Algorithm 2.7.4 is suitably chosen, α will be a root of a polynomial in $\mathbb{Z}[X]$ of some degree $k < n$, and this polynomial will probably be discovered by Algorithm 2.7.4. Of course, the results of Algorithm 2.7.4 may not correspond to exact relations, so to be sure that one has found a factor, one must algebraically divide A by its tentative divisor.

Although this method does not seem very clean and rigorous, it is certainly the easiest to implement. Hence, it should perhaps be tried before any of the more sophisticated methods above. In fact, in [LLL], it is shown how to make this method into a completely rigorous method. (They use p-adic factors instead of complex roots, but the result is the same.)

3.6 Additional Polynomial Algorithms

3.6.1 Modular Methods for Computing GCD's in $\mathbb{Z}[X]$

Using methods inspired from the factoring methods over $\mathbb{Z}$, one can return to the problem of computing GCD's over the specific UFD $\mathbb{Z}$, and obtain an algorithm which can be faster than the algorithms that we have already seen. The idea is as follows. Let $D = (A, B)$ in $\mathbb{Z}[X]$, and let $Q = (A, B)$ in $\mathbb{F}_p[X]$ where Q is monic. Then $D \bmod p$ is a common divisor of A and B in $\mathbb{F}_p[X]$, hence D divides Q in the ring $\mathbb{F}_p[X]$. (We should put $\bar{}$ to distinguish polynomials in $\mathbb{Z}[X]$ from polynomials in $\mathbb{F}_p[X]$, but the language makes it clear.)

If p does not divide both $\ell(A)$ and $\ell(B)$, then p does not divide $\ell(D)$ and so $\deg(D) \leq \deg(Q)$. If, for example, we find that $Q = 1$ in $\mathbb{F}_p[X]$, it follows that D is constant, hence that $D = (\mathrm{cont}(A), \mathrm{cont}(B))$. This is in general much easier to check than to use any version of the Euclidean algorithm over a UFD (Algorithm 3.3.1 for example). Note also that, contrary to the case of integers, two random polynomials over $\mathbb{Z}$ are in general coprime. (In fact a single random polynomial is in general irreducible.) In general however, we are in a non-random situation so we must work harder. Assume without loss of generality that A and B are primitive.

So as not to be bothered with leading coefficients, instead of D, we will compute an integer multiple $D_1 = c \cdot (A, B)$ such that

$$\ell(D_1) = (\ell(A), \ell(B)),$$

(i.e. with $c = \ell(D)/(\ell(A), \ell(B))$). We can then recover $D = \mathrm{pp}(D_1)$ since we have assumed A and B primitive.

Let M be the smallest of the bounds given by Theorem 3.5.1 for the two polynomials ℓA and ℓB, where $\ell = (\ell(A), \ell(B))$, and where we limit the degree of the factor by $\deg(Q)$. Assume for the moment that we skip the Hensel step, i.e. that we take $p > 2M$ (which in any case is the best choice if this leaves p in single precision). Compute the unique polynomial $Q_1 \in \mathbb{Z}[X]$ such that $Q_1 \equiv \ell Q \pmod{p}$ and having all its coefficients in $[-\frac{1}{2}p, \frac{1}{2}p[$. If $\mathrm{pp}(Q_1)$ is a common divisor of A and B (in $\mathbb{Z}[X]$!), then since D divides $Q \bmod p$, it follows that $(A, B) = \mathrm{pp}(Q_1)$. If it is not a common divisor, it is not difficult to see that this will happen only if p divides the leading term of one of the intermediate polynomials computed in the primitive form of Euclid's algorithm over a UFD (Algorithm 3.2.10), hence this will not occur often. If this phenomenon occurs, try again with another prime, and it should quickly work.

If M is really large, then one can use Hensel-type methods to determine $D_1 \bmod p^e$ for sufficiently large e. The techniques are completely analogous to the ones given in the preceding sections and are left to the reader.

Perhaps the best conclusion for this section is to quote Knuth essentially verbatim:

"The GCD algorithms sketched here are significantly faster than those of Sections 3.2 and 3.3 except when the polynomial remainder sequence is very short. Perhaps the best general procedure would be to start with the computation of (A, B) modulo a fairly small prime p, not a divisor of both $\ell(A)$ and $\ell(B)$. If the result Q is 1, we are done; if it has high degree, we use Algorithm 3.3.1; otherwise we use one of the above methods, first computing a bound for the coefficients of D_1 based on the coefficients of A and B and on the (small) degree of Q. As in the factorization problem, we should apply this procedure to the reverses of A and B and reverse the result, if the trailing coefficients are simpler than the leading ones."

3.6.2 Factorization of Polynomials over a Number Field

This short section belongs naturally in this chapter but uses notions which are introduced only in Chapter 4, so please read Chapter 4 first before reading this section if you are not familiar with number fields.

In several instances, we will need to factor polynomials not only over $\mathbb{Q}$ but also over number fields $K = \mathbb{Q}(\theta)$. Following [Poh-Zas], we give an algorithm for performing this task (see also [Tra]).

Let $A(X) = \sum_{0 \le i \le m} a_i X^i \in K[X]$ be a non-zero polynomial. As usual, we can start by computing $A/(A, A')$ so we can transform it into a squarefree polynomial, since $K[X]$ is a Euclidean domain. On the other hand, note that it is not always possible to compute the content of A since the ring of integers $\mathbb{Z}_K$ of K is not always a PID.

Call σ_j the $m = [K : \mathbb{Q}]$ embeddings of K into $\mathbb{C}$. We can extend σ_j naturally to $K[X]$ by acting on the coefficients, and in particular we can define the *norm* of A as follows

$$\mathcal{N}(A) = \prod_{1 \le j \le m} \sigma_j(A),$$

and it is clear by Galois theory that $\mathcal{N}(A) \in \mathbb{Q}[X]$.

We have the following lemmas. Note that when we talk of factorizations of polynomials, it is always up to multiplication by units of $K[X]$, i.e. by elements of K.

Lemma 3.6.1. *If $A(X) \in K[X]$ is irreducible then $\mathcal{N}(A)(X)$ is equal to the power of an irreducible polynomial of $\mathbb{Q}[X]$.*

Proof. Let $\mathcal{N}(A) = \prod_i N_i^{e_i}$ be a factorization of $\mathcal{N}(A)$ into irreducible factors in $\mathbb{Q}[X]$. Since $A \mid \mathcal{N}(A)$ in $K[X]$ and A is irreducible in $K[X]$, we have $A \mid N_i$ in $K[X]$ for some i. But since $N_i \in \mathbb{Q}[X]$, it follows that $\sigma_j(A) \mid N_i$ for all j, and consequently $\mathcal{N}(A) \mid N_i^m$ in $K[X]$, hence in $\mathbb{Q}[X]$, so $\mathcal{N}(A) = N_i^{m'}$ for some $m' \le m$. □

Lemma 3.6.2. *Let $A \in K[X]$ be a squarefree polynomial, where $K = \mathbb{Q}(\theta)$. Then there exists only a finite number of $k \in \mathbb{Q}$ such that $\mathcal{N}(A(X - k\theta))$ is not squarefree.*

Proof. Denote by $(\beta_{i,j})_{1 \le i \le m}$ the roots of $\sigma_j(A)$. If $k \in \mathbb{Q}$, it is clear that $\mathcal{N}(A(X - k\theta))$ is not squarefree if and only if there exists i_1, i_2, j_1, j_2 such that

$$\beta_{i_1,j_1} + k\sigma_{j_1}(\theta) = \beta_{i_2,j_2} + k\sigma_{j_2}(\theta),$$

or equivalently $k = (\beta_{i_1,j_1} - \beta_{i_2,j_2})/(\sigma_{j_2}(\theta) - \sigma_{j_1}(\theta))$ and there are only a finite number of such k. □

The following lemma now gives us the desired factorization of A in $K[X]$.

Lemma 3.6.3. *Assume that $A(X) \in K[X]$ and $\mathcal{N}(A)(X) \in \mathbb{Q}[X]$ are both squarefree. Let $\mathcal{N}(A) = \prod_{1 \le i \le g} N_i$ be the factorization of $\mathcal{N}(A)$ into irreducible factors in $\mathbb{Q}[X]$. Then $A = \prod_{1 \le i \le g} \gcd(A, N_i)$ is a factorization of A into irreducible factors in $K[X]$.*

Proof. Let $A = \prod_{1 \le i \le h} A_i$ be the factorization of A into irreducible factors in $K[X]$. Since $\mathcal{N}(A)$ is squarefree, $\mathcal{N}(A_i)$ also hence by Lemma 3.6.1 $\mathcal{N}(A_i) = N_{j(i)}$ for some $j(i)$. Furthermore since for $j \ne i$, $\mathcal{N}(A_i A_j) \mid \mathcal{N}(A)$ hence is squarefree, $\mathcal{N}(A_i)$ is coprime to $\mathcal{N}(A_j)$. So by suitable reordering, we obtain $\mathcal{N}(A_i) = N_i$ and also $g = h$. Finally, since for $j \ne i$, A_j is coprime to N_i it

follows that $A_i = \gcd(A, N_i)$ in $K[X]$ (as usual up to multiplicative constants), and the lemma follows. □

With these lemmas, it is now easy to give an algorithm for the factorization of $A \in K[X]$.

Algorithm 3.6.4 (Polynomial Factorization over Number Fields). Let $K = \mathbb{Q}(\theta)$ be a number field, $T \in \mathbb{Q}[X]$ the minimal monic polynomial of θ. Let $A(X)$ be a non-zero polynomial in $K[X]$. This algorithm finds a complete factorization of A in $K[X]$.

1. [Reduce to squarefree] Set $U \leftarrow A/(A, A')$ where (A, A') is computed in $K[X]$ using the sub-resultant Algorithm 3.3.1. (Now U will be a squarefree primitive polynomial. In this step, we could also use the squarefree decomposition Algorithm 3.4.2 to reduce still further the degree of U).
2. [Initialize search] Let $U(X) = \sum_{0 \le i \le m} u_i X^i \in K[X]$ and write $u_i = g_i(\theta)$ for some polynomial $g_i \in \mathbb{Q}[X]$. Set $G(X, Y) \leftarrow \sum_{0 \le i \le m} g_i(Y) X^i \in \mathbb{Q}[X, Y]$ and $k \leftarrow 0$.
3. [Search for squarefree norm] Using the sub-resultant Algorithm 3.3.7 over the UFD $\mathbb{Q}[Y]$, compute $N(X) \leftarrow R_Y(T(Y), G(X - kY, Y))$ where R_Y denotes the resultant with respect to the variable Y. If $N(X)$ is not squarefree (tested using Algorithm 3.3.1), set $k \leftarrow k + 1$ and go to step 3.
4. [Factor norm] (Here $N(X)$ is squarefree) Using Algorithm 3.5.7, let $N \leftarrow \prod_{1 \le i \le g} N_i$ be a factorization of N in $\mathbb{Q}[X]$.
5. [Output factorization] For $i = 1, \ldots, g$ set $A_i(X) \leftarrow \gcd(U(X), N_i(X + k\theta))$ computed in $K[X]$ using Algorithm 3.3.1, output A_i and the exponent of A_i in A (obtained simply by replacing A by A/A_i as long as $A_i \mid A$). Terminate the algorithm.

Proof. The lemmas that we have given above essentially prove the validity of this algorithm, apart from the easily checked fact that the sub-resultant computed in step 3 indeed gives the norm of the polynomial U. □

Remarks.

(1) The norm of U could also be computed using floating point approximations to the roots of T, since (if our polynomials have algebraic integer coefficients) it will have coefficients in $\mathbb{Z}$. This is often faster than sub-resultant computations, but requires careful error bounds.

(2) Looking at the proof of Lemma 3.6.2, it is also clear that floating point computations allow us to give the list of values of k to avoid in step 3, so no trial and error is necessary. However this is not really important since step 3 is in practice executed only once or twice.

(3) The factors that we have found are not necessarily in $\mathbb{Z}_K[X]$, and, as already mentioned, factoring in $\mathbb{Z}_K[X]$ requires a little extra work since $\mathbb{Z}_K$ is not necessarily a PID.

3.6.3 A Root Finding Algorithm over $\mathbb{C}$

In many situations, it is useful to compute explicitly, to some desired approximation, all the complex roots of a polynomial. There exist many methods for doing this. It is a difficult problem of numerical analysis and it is not my intention to give a complete description here, or even to give a description of the "best" method if there is one such. I want to give one reasonably simple algorithm which works most of the time quite well, although it may fail in some situations. In practice, it is quite sufficient, especially if one uses a multi-precision package which allows you to increase the precision in case of failure.

This method is based on the following proposition.

Proposition 3.6.5. *If $P(X) \in \mathbb{C}[X]$ and $x \in \mathbb{C}$, then if $P(x) \neq 0$ and $P'(x) \neq 0$ there exists a positive real number λ such that*

$$\left|P\left(x - \lambda \frac{P(x)}{P'(x)}\right)\right| < |P(x)|.$$

Proof. Trivial by Taylor's theorem. In fact, this proposition is valid for any analytic function in the neighborhood of x, and not only for polynomials. □

Note also that as soon as x is sufficiently close to a simple root of P, we can take $\lambda = 1$, and then the formula is nothing but Newton's formula, and as usual the speed of convergence is quadratic.

This leads to the following algorithm, which I call Newton's modified algorithm. Since we will be using this algorithm for irreducible polynomials over $\mathbb{Q}$, we can assume that the polynomial we are dealing with is at least squarefree. The modifications necessary to handle the general case are easy and left to the reader.

Algorithm 3.6.6 (Complex Roots). Given a squarefree polynomial P, this algorithm outputs its complex roots (in a random order). In quite rare cases the algorithm may fail. On the other hand it is absolutely necessary that the polynomial be squarefree (this can be achieved by replacing P by $P/(P, P')$).

1. [Initializations] Set $Q \leftarrow P$, compute P', set $Q' \leftarrow P'$, and set $n \leftarrow \deg(P)$. Finally, set $f \leftarrow 1$ if P has real coefficients, otherwise set $f \leftarrow 0$.
2. [Initialize root finding] Set $x \leftarrow 1.3 + 0.314159i$, $v \leftarrow Q(x)$ and $m \leftarrow |v|^2$.
3. [Initialize recursion] Set $c \leftarrow 0$ and $dx \leftarrow v/Q'(x)$. If $|dx|$ is smaller than the desired absolute accuracy, go to step 5.
4. [Try a λ] Set $y \leftarrow x - dx$, $v_1 \leftarrow Q(y)$ and $m_1 \leftarrow |v_1|^2$. If $m_1 < m$, set $x \leftarrow y$, $v \leftarrow v_1$, $m \leftarrow m_1$ and go to step 3. Otherwise, set $c \leftarrow c + 1$, $dx \leftarrow dx/4$. If $c < 20$ go to step 4, otherwise output an error message saying that the algorithm has failed.

5. [Polish root] Set $x \leftarrow x - P(x)/P'(x)$ twice.
6. [Divide] If $f = 0$ or if $f = 1$ and the absolute value of the imaginary part of x is less than the required accuracy, set it equal to 0, output x, set $Q(X) \leftarrow Q(X)/(X - x)$ and $n \leftarrow n - 1$. Otherwise, output x and $\bar{x}$, set $Q(X) \leftarrow Q(X)/(X^2 - 2\,\mathrm{Re}(x)X + |x|^2)$ and $n \leftarrow n - 2$. Finally, if $n > 0$ then go to step 2, otherwise terminate the algorithm.

Remarks.

(1) The starting value $1.3 + 0.314159i$ given in step 2 is quite arbitrary. It has been chosen so as not to be too close to a trivial algebraic number, and not too far from the real axis, although not exactly on it.

(2) The value 20 taken in step 4, as well as the division by 4, are also arbitrary but correspond to realistic situations. If we find $m_1 \geq m$, this means that we are quite far away from the "attraction zone" of a root. Hence, thanks to Proposition 3.6.5, it is preferable to divide the increment by 4 and not by 2 for example, so as to have a much higher chance of winning next time. Similarly, the limitation of 20 correspond to an increment which is $4^{20} \approx 10^{12}$ times smaller than the Newton increment, and this is in general too small to make any difference. In that case, it will be necessary to increase the working precision.

(3) After each division done in step 6, the quality of the coefficients of Q will deteriorate. Hence, after finding an approximate root, it is essential to polish it, using for example the standard Newton iteration, but with the polynomial P and not Q. It is not necessary to use a factor λ since we are in principle well inside the attraction zone of a root. Two polishing passes will, in principle, be enough.

(4) The divisions in step 6 are simple to perform. If $Q(X) = \sum_{0 \leq i \leq n} q_i X^i$ and $A(X) = \sum_{0 \leq i \leq n-1} a_i X^i = Q(X)/(X - x)$, then set $a_{n-1} \leftarrow q_n$ and for $i = n - 1, \ldots, i = 1$ set $a_{i-1} \leftarrow q_i + x a_i$. Similarly, if $B(X) = \sum_{0 \leq i \leq n-2} b_i X^i = Q(X)/(X^2 - \alpha X + \beta)$, then set $b_{n-2} \leftarrow q_n$, $b_{n-3} \leftarrow q_{n-1} + \alpha b_{n-2}$ and for $i = n - 2, \ldots, i = 2$ set $b_{i-2} \leftarrow q_i + \alpha b_{i-1} - \beta b_i$.

(5) Instead of starting with $\lambda = 1$ as coefficient of $Q(x)/Q'(x)$ in step 3, it may be better to start with

$$\lambda = \min\left(1, \frac{2|Q'(x)|^2}{|Q(x)||Q''(x)|}\right).$$

This value is obtained by looking at the error term in the Taylor expansion proof of Proposition 3.6.5. If this value is too small, then we are probably going to fail, and in fact x is converging to a root of $Q'(X)$ instead of $Q(X)$. If this is detected, the best solution is probably to start again in step 2 with a different starting value. This of course can also be done when $c = 20$ in step 4. We must however beware of doing this too systematically, for failure may indicate that the coefficients of the polynomial P are ill conditioned, and in that case the best remedy is to modify the coefficients

of P by a suitable change of variable (typically of the form $X \mapsto aX$). It must be kept in mind that for ill conditioned polynomials, a very small variation of a coefficient can have a drastic effect on the roots.

(6) In step 6, instead going back to step 2 if $n > 0$, we can go back only if $n > 2$, and treat the cases $n = 1$ and $n = 2$ by using the standard formulas. Care must then be taken to polish the roots thus obtained, as is done in step 5.

3.7 Exercises for Chapter 3

1. Write an algorithm for multiplying two polynomials, implicitly based on a recursive use of the splitting formulas explained in Section 3.1.2.

2. Let P be a polynomial. Write an algorithm which computes the coefficients of the polynomial $P(X+1)$ without using an auxiliary array or polynomial.

3. Let K be a commutative ring which is not necessarily a field. It has been mentioned after Algorithm 3.1.1 that the Euclidean division of A by B is still possible in $K[X]$ if the leading coefficient $\ell(B)$ is invertible in K. Write an algorithm performing this Euclidean division after multiplying A and B by the inverse of $\ell(B)$, and compare the performance of this algorithm with the direct use of Algorithm 3.1.1 in the case $K = \mathbb{Z}/r\mathbb{Z}$.

4. Modify Algorithm 3.3.1 so that A and B are divided by their respective contents every 10 iterations. Experiment and convince yourself that this modification leads to polynomials A and B having much larger coefficients later on in the Algorithm, hence that this is a bad idea.

5. Write an extended version of Algorithm 3.3.1 which computes not only (A, B) but also U and V such that $AU + BV = r \cdot (A, B)$ where r is a non-zero constant (Hint: add a fourth variable in Algorithm 1.3.6 to take care of r). Show that when $(A, B) = 1$ this can always be done with r equal to the resultant of A and B.

6. Show that if A, B and C are irreducible polynomials over a UFD R and if C divides AB but is not a unit multiple of A, then C divides B (Hint: use the preceding exercise). Deduce from this that $R[X]$ is a UFD.

7. Using for example the sub-resultant algorithm, compute explicitly the discriminant of the trinomials $X^3 + aX + b$ and $X^4 + aX + b$. Try to find the general formula for the discriminant of $X^n + aX + b$.

8. Call R_i the i-th row of Sylvester's determinant, for $1 \leq i \leq n+m$. Show that if we replace for all $1 \leq i \leq n$ simultaneously R_i by

$$\sum_{k=0}^{i-1} (b_k R_{i-k} - a_k R_{i+m-k})$$

and then suppress the last m rows and columns of the resulting matrix, the $n \times n$ determinant thus obtained is equal to the determinant of Sylvester's matrix.

9. If $Q(X) = (X - a)P(X)$, compute the discriminant of Q in terms of a and of the discriminant of P.

10. Show how to modify the sub-resultant Algorithm 3.3.7 so that it can compute correctly when the coefficients of the polynomials are for example polynomials (in another variable) with real coefficients.

11. Show the following result, due to Eisenstein: if p is prime and $A(X) = \sum_{0 \leq i \leq n} a_i X^i$ is a polynomial in $\mathbb{Z}[X]$ such that $p \nmid a_n$, $p \mid a_i$ for all $i < n$ and $p^2 \nmid a_0$, then A is irreducible in $\mathbb{Z}[X]$.

12. Using the ideas of Section 3.4, write an algorithm to compute the square root of a mod p, or to determine whether none exist. Implement your algorithm and compare it with Shanks's Algorithm 1.5.1.

13. Using the Möbius inversion formula (see [H-W] Section 16.4) show that the number of monic irreducible polynomials of degree n over $\mathbb{F}_p$ is equal to

$$\frac{1}{n} \sum_{d|n} \mu\left(\frac{n}{d}\right) p^d$$

where $\mu(n)$ is the Möbius function (i.e. 0 if n is not squarefree, and equal to $(-1)^k$ if n is a product of k distinct prime factors).

14. Extend Proposition 3.4.7 and Algorithm 3.4.8 to general prime numbers p, using $U_p(X) = X + X^p + \cdots + X^{p^{d-1}}$. Compare in practice the expected speed of the resulting algorithm to that of Algorithm 3.4.6.

15. Show that, as claimed in the proof of Algorithm 3.4.8, the polynomial U has exactly 2^{d-1} roots in $\mathbb{F}_{2^d}$.

16. Generalizing the methods of Section 3.4, write an algorithm to factor polynomials in $\mathbb{F}_q[X]$, where $q = p^d$ and $\mathbb{F}_q$ is given by an irreducible polynomial of degree d in $\mathbb{F}_p[X]$.

17. Let $B(X) \in \mathbb{F}_p[X]$ be a squarefree polynomial with r distinct irreducible factors. Show that if $T(X)$ is a polynomial corresponding to a randomly chosen element of the kernel obtained in step 2 of Algorithm 3.4.10 and if $p \geq 3$, the probability that $(B(X), T(X)^{(p-1)/2} - 1)$ gives a non-trivial factor of B is greater than or equal to $4/9$.

18. Let K be any field, $a \in K$ and p a prime number. Show that the polynomial $X^p - a$ is reducible in $K[X]$ if and only if it has a root in K. Generalize to the polynomials $X^{p^r} - a$.

19. Let p be an odd prime and q a prime divisor of $p-1$. Let $a \in \mathbb{Z}$ be a primitive root modulo p. Using the preceding exercise, show that for any $k \geq 1$ the polynomial

$$X^q + pX^k - a$$

is irreducible in $\mathbb{Q}[X]$.

20. Let p and q be two odd prime numbers. We assume that $q \equiv 2 \pmod 3$ and that p is a primitive root modulo q (i.e. that $p \bmod q$ generates $(\mathbb{Z}/q\mathbb{Z})^*$). Show that the polynomial

$$X^{q+1} - X + p$$

is irreducible in $\mathbb{Q}[X]$. (Hint: reduce mod p and mod 2.)

21. Separating even and odd powers, any polynomial A can be written in the form $A(X) = A_0(X^2) + XA_1(X^2)$. Set $T(A)(X) = A_0(X)^2 - XA_1(X)^2$. With the notations of Theorem 3.5.1, show that for any k

$$|b_j| \leq \binom{n-1}{j} |T^k(A)|^{1/2^k} + \binom{n-1}{j-1} |a_m|.$$

What is the behavior of the sequence $|T^k(A)|^{1/2^k}$ as k increases?

22. In Algorithms 3.5.5 and 3.5.6, assume that $p = q$, that A and B are monic, and set $D = AU$, $D_1 = A_1U_1$, $E = BV$, $E_1 = B_1V_1$. Denote by (C, p^2) the ideal of $\mathbb{Z}[X]$ generated by $C(X)$ and p^2. Show that

$$D_1 \equiv 3D^2 - 2D^3 \pmod{(C,p^2)} \quad \text{and} \quad E_1 \equiv 3E^2 - 2E^3 \pmod{(C,p^2)}.$$

Then show that A_1 (resp. B_1) is the monic polynomial of the lowest degree such that $E_1A_1 \equiv 0 \pmod{(C,p^2)}$ (resp. $D_1B_1 \equiv 0 \pmod{(C,p^2)}$).

23. Write a general algorithm for finding all the roots of a polynomial in $\mathbb{Q}_p$ to a given p-adic precision, using Hensel's lemma. Note that multiple roots at the mod p level create special problems which have to be treated in detail.

24. Denote by $(\ ,\ )_p$ the GCD taken over $\mathbb{F}_p[X]$. Following Weinberger, Knuth asserts that if $A \in \mathbb{Z}[X]$ is a product of exactly k irreducible factors in $\mathbb{Z}[X]$ (not counting multiplicity) then

$$\lim_{x\to\infty} \frac{\sum_{p\leq x} \deg(X^p - X, A(X))_p}{\sum_{p\leq x} 1} = k.$$

Explore this formula as a heuristic method for determining the irreducibility of a polynomial over $\mathbb{Z}$.

25. Find the complete decomposition into irreducible factors of the polynomial $X^4 + 1$ modulo every prime p using the quadratic reciprocity law and the identities given in Section 3.5.2.

26. Discuss the possibility of computing polynomial GCD's over $\mathbb{Z}$ by computing GCD's of *values* of the polynomials at suitable points. (see [Schön]).

27. Using the ideas of Section 3.4.2, modify the root finding Algorithm 3.6.6 so that it finds the roots of a any polynomial, squarefree or not, with their order of multiplicity. For this question to make practical sense, you can assume that the polynomial has integer coefficients.

28. Let $P(X) = X^3 + aX^2 + bX + c \in \mathbb{R}[X]$ be a monic squarefree polynomial. Let θ_i $(1 \leq i \leq 3)$ be the roots of P in $\mathbb{C}$ and let

$$\alpha_1 = (\theta_1 + \rho^2\theta_2 + \rho\theta_3)^3, \quad \alpha_2 = (\theta_1 + \rho\theta_2 + \rho^2\theta_3)^3.$$

Let $A(X) = (X - \alpha_1)(X - \alpha_2)$.

a) Compute explicitly the coefficients of $A(X)$.

b) Show that $-27\operatorname{disc}(P) = \operatorname{disc}(A)$, and give an expression for this discriminant.

c) Show how to compute the roots of P knowing the roots of A.

29. Let $P(X) = X^4 + aX^3 + bX^2 + cX + d \in \mathbb{R}[X]$ be a monic squarefree polynomial. Let θ_i $(1 \leq i \leq 4)$ be the roots of P in $\mathbb{C}$, and let

$$\alpha_1 = (\theta_1 + \theta_2)(\theta_3 + \theta_4) \quad \alpha_2 = (\theta_1 + \theta_3)(\theta_2 + \theta_4) \quad \alpha_3 = (\theta_1 + \theta_4)(\theta_2 + \theta_3),$$

and

$$\beta_1 = \theta_1\theta_2 + \theta_3\theta_4 \quad \beta_2 = \theta_1\theta_3 + \theta_2\theta_4 \quad \beta_3 = \theta_1\theta_4 + \theta_2\theta_3\,.$$

Finally, let $A(X) = (X - \alpha_1)(X - \alpha_2)(X - \alpha_3)$ and $B(X) = (X - \beta_1)(X - \beta_2)(X - \beta_3)$.

a) Compute explicitly the coefficients of $A(X)$ and $B(X)$ in terms of those of $P(X)$.

b) Show that $\operatorname{disc}(P) = \operatorname{disc}(A) = \operatorname{disc}(B)$, and give an expression for this discriminant.

c) Show how to compute the roots of P knowing the roots of A.

30. Recall that the *first case* of Fermat's last "theorem" (FLT) states that if l is an odd prime, then $x^l + y^l + z^l = 0$ implies that $l \mid xyz$. Using elementary arguments (i.e. no algebraic number theory), it is not too difficult to prove the following theorem, essentially due to Sophie Germain.

Theorem 3.7.1. *Let l be an odd prime, and assume that there exists an integer k such that $k \equiv \pm 2 \pmod 6$, $p = lk + 1$ is prime and $p \nmid (k^k - 1)W_k$ where W_k is the resultant of the polynomials $X^k - 1$ and $(X + 1)^k - 1$. Then the first case of FLT is true for the exponent l.*

It is therefore important to compute W_k and in particular its prime factors. Give several algorithms for doing this, and compare their efficiency. Some familiarity with number fields and in particular with cyclotomic fields is needed here.

31. Let $A(X) = a_nX^n + \cdots + a_1X + a_0$ be a polynomial, with $a_n \neq 0$. Show that for any positive integer k,

$$\operatorname{disc}(A(X^k)) = (-1)^{nk(k+3)/2}k^{nk}(a_na_0)^{k-1}\operatorname{disc}(A)^k\,.$$