Exploring Cryptography Using the Sage Computer Algebra System

by

Minh Van Nguyen

Thesis submitted

in partial fulfillment of the Requirements for the Degree of

Bachelor of Science (Honours) in Computer Science

Supervisor: Dr Alasdair McAndrew

School of Engineering and Science

Victoria University

December, 2009

Exploring Cryptography Using the Sage Computer Algebra System

Declaration

I hereby declare that this submission is my own work and to the best of my knowledge it contains no material previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any other degree or diploma at Victoria University or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by colleagues, with whom I have worked at Victoria University or elsewhere, during my candidature, is fully acknowledged.

I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Minh Van Nguyen
05 December 2009

Exploring Cryptography Using the Sage Computer Algebra System

Minh Van Nguyen
nguyenminh2@gmail.com
Victoria University, 2009

Supervisor: Dr Alasdair McAndrew
Alasdair.McAndrew@vu.edu.au

Abstract

Cryptography has become indispensable in areas such as e-commerce, the legal safeguarding of medical records, and secure electronic communication. Hence, it is incumbent upon software engineers to understand the concepts and techniques underlying the cryptosystems that they implement. An educator needs to consider which topics to cover in a course on cryptography as well as how to present the concepts and techniques to be covered in the course. This thesis contributes to the field of cryptography pedagogy by discussing and implementing small-scale cryptosystems whose encryption and decryption processes can be stepped through by hand. Our implementation has been accepted and integrated into the code base of the computer algebra system Sage. As Sage is free and open source, students and educators of cryptology need not worry about paying license fees in order to use Sage, but can instead concentrate on exploring cryptography using Sage's built-in support for cryptography.

Acknowledgements

I am indebted to several people whose support, inspiration and encouragement have been invaluable during the course of writing and performing the work described in the thesis. First and foremost is my thesis supervisor Dr Alasdair McAndrew who provided a stimulating environment during the time that we worked together. His probing questions and indefatigable support have contributed to enhancing the quality of the thesis.

The majority of my software development effort took place on the compute node `sage.math` [101], which is one of four machines comprising the Sage cluster and supported by US National Science Foundation Grant No. DMS-0821725. I wish to extend my gratitude to Associate Professor William A. Stein of the University of Washington, USA, for allowing me access to the Sage cluster in order to develop and test all of the software implementation described in this thesis. Professor Stein is a technical reviewer of my implementation of S-DES [83]. He also provided me with access to the machine `bsd.math.washington.edu` and expended considerable effort so that I have access to the computer network SkyNet, which is a research network of the US Department of Defense and administered by Mariah Lenox. Software testing conducted on the Sage cluster, the machine `bsd.math.washington.edu`, and SkyNet has contributed to enhancing the quality, stability and portability of my enhancements to the cryptography module of the Sage computer algebra system.

Martin Albrecht of the University of London, UK, is a technical reviewer of my implementation of S-DES [83] and the sole technical reviewer of my implementation of Mini-AES [80]. His constructive and timely feedback played a considerable role in enhancing the quality of those two implementations. He also read an early draft of this thesis and made numerous suggestions to clarify the exposition of Chapters 1 and 2. I also wish to thank Mr Albrecht for reminding me that Sage distributes the PyCrypto library. This observation has contributed to improving the exposition of Chapter 2.

Nick Alexander of the University of California at Irvine, USA, is the sole technical reviewer of my patch at ticket `#6222` [85]. Sage developer and release manager Mike Hansen reviewed my implementation [79] of an algorithm for solving the subset sum problem over super-increasing sequences. He is also the sole technical reviewer of my patch at ticket `#6176` [81], and co-reviewed my patches at ticket `#7123` [78].

Prior to starting development of the Sage implementation of cryptosystems described in Chapters 3 to 6, I submitted a patch to ticket `#5529` [77] in order to enhance the documentation of the Sage cryptography module. Associate Professor

John Palmieri of the University of Washington, USA, is the technical reviewer of that patch. His comments and reviewer patch for ticket #5529 have contributed to improving the overall quality of the documentation for the Sage cryptography module.

I wish to acknowledge Professor Robert A. Beezer of the University of Puget Sound, USA, for bringing to my attention both the chi-square and squared-differences statistical measures and our subsequent discussion on using those measures for cryptanalysis of the shift and affine cryptosystems. Professor Beezer is the technical reviewer of my implementation of the shift [84, 82, 78] and affine [76] cryptosystems. His feedback have contributed to considerably enhancing my original implementations. Professor Beezer also reviewed a draft of the thesis and provided comments on typographical and stylistic errors.

I would like to extend my gratitude to Professor Bernhard Esslinger, leader of the CrypTool [37] project, for inviting me to join the project's documentation team. Since joining the documentation team of CrypTool, I have had ample opportunities to enhance the CrypTool tutorial with Sage code for learning cryptography. Some of the cryptography implementations described in the thesis have made their way into the CrypTool tutorial as examples showing the working of particular cryptosystems. Professor Esslinger also made numerous comments on a draft of the thesis that help to clarify many issues.

Finally, I wish to thank Brett Robertson of Victoria University, Australia, and two anonymous reviewers for reading a draft of the thesis. Mr Robertson made numerous comments that help to improve the organization of the thesis and the exposition of Chapters 1 and 7. One of the two anonymous reviewers reminded me that Sage also supports the PyCrypto library, an observation which helps to improve my exposition of Chapter 2. The anonymous reviewers, and many people who read a draft of the thesis, pointed out numerous grammatical, spelling and stylistic errors. Any errors that remain are solely my responsibility.

Minh Van Nguyen

Victoria University
December 2009

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Cryptography is the science (and sometimes art) of secret writing in which the goal is to hide information. Originally the preserve of diplomats and military organizations, cryptography has become indispensable in areas such as e-commerce, the legal safeguarding of medical records, transmission of military orders, and even something seemingly routine as secure email communications. Imagine that we are composing a confidential email to someone. Having written the email, we can send it in one of two ways. The first, and usually convenient, way is to simply send the email and not care about how it would be delivered. Sending an email in this manner is similar to writing our confidential message on a postcard and post it without enclosing our postcard inside an envelope. Anyone who can access our postcard can see our message. On the other hand, we can scramble the confidential message in one way or another prior to sending the email. Scrambling our message is similar to enclosing our postcard inside an envelope. While this particular method of hiding our postcard message is not 100% secure, at least we know that anyone wanting to read our postcard has to open the envelope. The discipline of cryptography offers more secure techniques for hiding messages than enclosing messages inside an envelope.

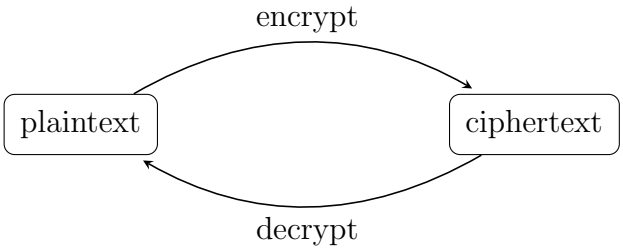

Figure 1.1: The encryption/decryption cycle.

In cryptography parlance, our message is referred to as *plaintext*. The process of scrambling our message using a *key* is called *encryption*. After encrypting our message, the scrambled version is called *ciphertext*. From the ciphertext, we can recover our original unscrambled message via a process known as *decryption*. Figure 1.1 illustrates the encryption and decryption processes. A *cryptosystem* is a combination of the encryption and decryption algorithms, whereas *cryptanalysis* is the science (and sometimes art) of undermining or "breaking" encryption schemes. Our discussion so far has excluded other aspects and subtleties of cryptography.

Readers who require an in-depth discussion of cryptography are referred to specialized texts such as Hoffstein et al. [48], Menezes et al. [67], Mollin [68], Stinson [113], and Trappe and Washington [114].

As we alluded to earlier, an encryption or decryption algorithm relies on a key, which may be a number or sequence of bits having some desirable properties. Cryptosystems can be distinguished based upon how they use their keys. A *secret-* or *symmetric-key* cryptosystem uses the same secret key for both encryption and decryption. In contrast, a *public-key* cryptosystem uses a public key to encrypt messages and decryption is performed using a corresponding private key. Public keys may be distributed or publicized, but for security reasons private keys are to be kept confidential.

No matter how strong or secure looking a cryptosystem is, it can be attacked or undermined in one way or another. In general, this statement does not hold true for the one-time pad, which is a prominent example of a cryptosystem that has been theoretically proven using results from information theory to be secure. Whether the one-time pad is practical or not is another issue. One method of assessing the strength of a cryptosystem is to study its robustness when subjected to all known attack techniques. Numerous classes of attacks can be mounted against a cryptosystem, with the most common technique being the *brute-force* attack, otherwise known as exhaustive key search: one simply decrypts a ciphertext using all the known keys of a cryptosystem until a key is found that results in some meaningful plaintext. Attack techniques come and go, but a vital time-honoured principle is what is known as *Kerckhoffs' principle*, named after Auguste Kerckhoffs von Nieuwenhof (1835–1903) who discussed it in 1883 in an essay entitled *La Cryptographie militaire*. Kerckhoffs' principle states that the security of a cryptosystem should not be based on any of its aspects but the secret key. That is, every aspect of a cryptosystem including its encryption and decryption routines are assumed to be well-known, but the problem generally is to determine the secret key.

1.1 Cryptography and computer security

Cryptography is one of many aspects of computer security. Choosing a secure cryptosystem is merely part of the work, as well as ensuring that the cryptosystem is implemented and applied in such a way as to not compromise its security. Cryptosystems abound that have been demonstrated to be strong and hence supposedly secure. However, if the implementation (whether that be hardware, software, or a combination of both) of a cryptosystem is flawed or vulnerable in some way, then by exploiting the flaw/vulnerability, an attack can be mounted against the cryptosystem. For practical guidelines on implementing a cryptosystem, refer to Ferguson and Schneier [38], or Schneier [104].

The use of a cryptosystem is not a guarantee of system security. Bruce Schneier is often credited with the phrase, "Security is a process, not a product." By this, he means that cryptography is not some magic dust that one could sprinkle on a computer system and thus the system is secured. Rather, system security is a combination of tools, techniques, and strategies all operating within a specific context. These ideas are elaborated in further details with numerous examples in Schneier [105].

1.2 Thesis outline

This section presents an outline of the remainder of the thesis and our contribution to the field of cryptography pedagogy using a computer algebra system (CAS). Chapter 2 provides a description of CASs and outlines basic functionalities that all such systems support. Next, we outline topics that may be encountered in a course on cryptography, and review previous work on using CASs in cryptography education. After briefly touching upon various CASs, we discuss the relatively nascent free open source computer algebra system Sage and summarize the underlying philosophical principle that governs its existence and development model. In order to integrate CASs into a cryptography curriculum, one requires an understanding of functionalities specific to cryptography education that any CAS needs to support. Such a list of fundamental functionalities are presented, based upon which we compare and contrast the general purpose computer algebra systems FriCAS, Maple, Mathematica, Matlab, Maxima and Sage.

Sage is the primary focus of the thesis. Through our survey of CASs with respect to their support for cryptography education, we identify missing functionalities in Sage and set out to fill in a number of those functionalities. Chapter 3 describes the shift cryptosystem, one of many cryptosystems we have identified as missing built-in support in Sage. The chapter begins with some results from elementary number theory that lay the mathematical foundation of the shift cryptosystem. The mathematical concepts are then tied together to define the encryption and decryption functions, as well as to describe techniques for attacking this cryptosystem. The chapter concludes with numerous examples illustrating our Sage implementation of the shift cryptosystem.

A second cryptosystem for which Sage lacks built-in support is the affine cryptosystem. This cryptosystem is specified in Chapter 4, which generalizes the shift cryptosystem and continues the development of number theoretic techniques begun in Chapter 3. We then apply these mathematical concepts to analyze the key space of the affine cryptosystem, in addition to defining its encryption and decryption functions. All of the attack methods discussed in Chapter 3 can be brought to bear on the affine cryptosystem. Our implementation of this cryptosystem follows the same general interface as used by our implementation of the shift cryptosystem. Examples illustrating functionalities of our affine cryptosystem implementation are presented towards the end of Chapter 4.

Sage supports the full Data Encryption Standard (DES) through the PyCrypto third-party library, which is designed to provide cryptographic services instead of serving as a teaching tool. Chapter 5 describes a simplified version of DES known as S-DES that can be used as a teaching tool to introduce students of cryptology to the general structure of the full DES algorithm. The presentation includes the encryption and decryption functions of S-DES, as well as the permutation and round functions that comprise its foundation. We also provide numerous examples to illustrate functionalities of our Sage implementation of S-DES.

Our survey of computer algebra systems in Chapter 2 shows that Sage has built-in support for the Advanced Encryption Standard (AES) and some small-scale variants of the latter system. However, the existing support in Sage for AES and the latter's small-scale variants are designed as a framework for comparing different cryptanalytic techniques that can be brought to bear on the full AES algorithm. In order

to provide a simplified variant of AES that could be used in cryptography pedagogy, Chapter 6 describes a small-scale version of AES known as Mini-AES that allows cryptology students to manually step through the general structure of AES. Towards the end of Chapter 6, we also provide examples illustrating functionalities of our Sage implementation of Mini-AES.

Finally, Chapter 7 concludes the thesis and provides some directions for future research. Note that all of our enhancements to Sage as described in the thesis have been integrated into the Sage standard library and have undergone public peer reviews to ensure technical accuracy, completeness of documentation, and seamless integration into the Sage code base. The reference manual of our implementation is provided in Appendices A to E, while the source code of our implementation is available with the latest stable release of Sage, which as of this writing is Sage 4.2.1.

Chapter 2

A Survey of CAS for Cryptography Education

In this chapter, we survey a number of general purpose computer algebra systems for which a literature search reveals that those systems have been used for teaching and learning cryptography. The systems we shall consider are FriCAS, Maple, Mathematica, Matlab, Maxima and Sage. Our survey does not take into account special purpose software tools nor special purpose computer algebra systems such as Magma and Pari/GP. These latter two special purpose systems have efficient implementation of many algorithms required by various modern cryptosystems. However, to cover Magma and Pari/GP, or indeed any other special purpose software tools, would go beyond the scope of our investigation.

We begin in section 2.1 with a description of what computer algebra systems are, and various basic functionalities that all such systems support. Section 2.2 briefly touches upon topics that may be covered in a course on cryptography, and review previous work reported in the literature on using computer algebra systems in cryptography pedagogy. Throughout the section, we highlight the abundance use of closed source proprietary computer algebra systems in cryptography education, as compared to the paucity of literature reporting on the use of open source counterparts. Section 2.3 discusses the open source computer algebra system Sage, describes the underlying philosophical principle that governs its existence and development model, and survey cryptography research that has made use of functionalities of Sage. We next consider in section 2.4 functionalities specific to cryptography education that a computer algebra system needs to support. This is followed by a comparison of the above six systems with respect to cryptography related functionalities. Section 2.5 considers a case in which a lack of built-in support for a cryptosystem can be compensated for by using a computer algebra system's programming language to work through the encryption and decryption processes of the cryptosystem.

One of the conclusions that can be drawn from our survey is that Sage has support for a wider range of functionalities for teaching and learning cryptography than any other computer algebra systems considered in this chapter. Despite extensive software support for cryptography education, our survey also indicates that Sage lacks built-in support for various topics found in a course on cryptography. Section 2.6 identifies numerous missing features in Sage, and refers to various chapters

and appendices in the thesis that describe our work on filling in some of the missing functionalities.

2.1 Computer algebra systems

A computer algebra system (CAS) is a mathematics software package that is able to perform both symbolic and numerical mathematical computation. Among the early CASs were Macsyma [34, 50, 61] and REDUCE [45, 75], both of which are still in use and have only recently been distributed under open source licenses. We shall not attempt a definition of the term "open source", but interested readers are referred to the Open Source Initiative [90] for a discussion of the definition of "open source" as well as examples of open source software licenses. A modern descendant of Macsyma is Maxima [62]. From 1982 until 2001, William Schelter of The University of Texas at Austin maintained the Maxima branch of Macsyma. Starting from 1998, Maxima is licensed under the terms of the GNU General Public License (GPL) Version 2. As of January 2009, REDUCE [45] is covered by a modified Berkeley Software Distribution (BSD) license and the project is hosted on SourceForge.net. Modern general purpose, closed source proprietary CASs include Maple by Maplesoft, Matlab by The MathWorks, and Mathematica by Wolfram Research. Modern general purpose, open source CASs include Axiom (including its forks OpenAxiom and FriCAS), Maxima, Sage [112] and Yacas [94].

Irrespective of whether a CAS is open source or closed source, one may expect a CAS to support at minimum the following functionalities (this list is taken from [65]):

- Arbitrary precision arithmetic — Arithmetic over the integers, rationals, reals or complex numbers to any desired precision.
- Algebra of polynomials — Arithmetic of polynomials; factorization over the integers, rationals, reals, complex numbers or finite fields; simplification and partial fraction decomposition of rational functions.
- Calculus — Limits, derivatives, symbolic summation and product, definite and indefinite integration, and expansions of functions.
- Linear algebra — Solving systems of linear equations in both symbolic and numeric form, matrix algebra, determinants, eigenvalues and eigenvectors.
- Solution of non-linear equations — Solutions by radicals of all polynomials of degree less than five, and numerical solutions to any desired precision of systems of non-linear equations.
- Knowledge of functions — Support for transcendental functions and their properties.
- Graphics — Graphs of functions of one and two variables, parametric plots in two and three dimensions.

Many CASs support much more than the above bare-bone functionalities. Some CASs support all or a subset of the following functionalities:

- User interface — A command line interface; a graphical user interface.

- Animation — Animation of two- or three-dimensional graphics.
- Networking and grid computing — Support for parallel or distributed computation.
- Statistics — Standard probability distributions including binomial, Poisson, hypergeometric, normal, Student's t; mean, variance, correlation coefficients, chi-square test, regression.
- Transforms — Laplace, Fourier, fast Fourier, Mellin, and Z transforms.
- Solution of differential equations — Numerical or closed form solutions.
- Solution of difference equations — Closed form solutions of various types of difference equations.
- Special functions — Knowledge of transcendental functions including gamma and beta functions, Riemann's zeta function, hypergeometric functions. Functions arising from solutions of differential equations including Bessel, Airy, Mathieu, Legendre, Laguerre, Jacobi and elliptic functions. Knowledge of various classes of orthogonal polynomials.
- Number theory — Integer factorization, primality testing, prime number generation, greatest common divisor, least common multiple, quadratic and higher order residues; number theoretic functions such as $\varphi(n)$, $\pi(n)$, and $d(n)$; support for other functionalities arising from elementary number theory.
- Programming language — It should be possible to extend the functionalities of the CAS using a programming language.
- Typesetting — Format output in LaTeX for inclusion in documents.
- Other areas of mathematics — Specialized packages or functions for dealing with group theory, combinatorics, formal power series, geometry, topology, graph theory, mathematical optimization, algebraic number theory, etc.

2.2 CAS in cryptography education

Cryptography is a subject rich in both theory and applications, with a strong foundation in mathematics and at the same time whose influences can be felt in e-commerce and financial transactions. It is this richness in both theory and practice that allows cryptography to be taught from various perspectives. A course in cryptography might emphasize the mathematical foundation of classical and modern cryptosystems [48, 53, 92, 113], weave the historical foundation into the main discussion [13, 68], or be a generic computer security course [9, 41, 91, 103]. One could also blend theory with practice [100, 113] by incorporating special purpose software tools as a teaching aide [19, 26, 37, 92, 108]. On the surface, it seems there is a lack of consensus on a syllabus for an undergraduate course in cryptography. However, the SIGCSE recommendations for computer science curricula [4] include the following topics:

- Historical overview of cryptography
- Private-key cryptography and the key-exchange problem
- Public-key cryptography
- Digital signatures
- Security protocols
- Applications (zero-knowledge proofs, authentication, and so on)

Modern CASs, and their range of supported functionalities, open up many opportunities for incorporating them in mathematics and computer science education. Indeed, CASs have been used for teaching since their inception in the 1970s. For discussions on using CASs in mathematics education, see for example Buchberger [23, 24], Koepf [54], McAndrew [65], Monagan [69], Naismith and Sangwin [73], Pletsch [95], and Villate [117].

Since the mid-1990s, there has been some interest in using CASs for teaching cryptography. Baliga and Boztas [11] report their experience in using Maple for teaching introductory cryptography courses to advanced undergraduates in engineering as well as postgraduate students in information security. A CAS such as Maple allows students to explore non-trivial examples of encryption and decryption, bearing in mind that it is a teaching and learning tool and the student is not required to master the CAS. Understanding the limited support in Maple for the advanced algebra required by modern cryptography, Baliga and Boztas had also started using Magma [21], a specialized closed source CAS produced by the Computational Algebra Group of the University of Sydney for exploring problems in algebra, number theory, geometry and combinatorics. Cosgrave [30] also reports a similar investigation since the mid-1990s into using Maple for teaching number theory and cryptography. The cryptography topics covered are primarily seen as applications of elementary number theory, with special emphasis on modern cryptosystems and digital signature schemes based on number theory. Cosgrave's coverage of cryptography using a CAS does not cover classical cryptosystems, whereas Baliga and Boztas cover classical together with number theoretic cryptosystems. Eisenberg [35] discusses a mathematics course in which a student project is viewed as an application of linear algebra to cryptography. As part of a linear algebra course, students use Mathematica to implement Hill cipher [46, 47] encryption and decryption as well as cryptanalyzing the Hill cryptosystem.

Since the investigations of Baliga and Boztas [11], Cosgrave [30] and Eisenberg [35], there have been textbooks and specialized CAS packages that integrate the use of a CAS into a cryptography curriculum. The textbook by Trappe and Washington [114] covers both classical and modern cryptosystems. Of special note is its integration of Maple, Mathematica and Matlab into the topics covered. The authors use built-in commands of these CASs wherever possible and have also written custom commands [115] using the programming languages of those CASs to fill in various gaps. The textbook of Klima et al. [52] treats classical and modern cryptosystems as applications of algebra, and does not cover cryptography in as much depth as Trappe and Washington. Each cryptosystem presented is immediately followed by a discussion on using Maple and Matlab to carry out the encryption and

decryption procedures. Similar to Trappe and Washington, Klima et al. have also written custom commands in the programming languages of Maple and Matlab to fill in cryptography specific functionalities missing from those two CASs. May [63, 64] has developed a complete set of Maple worksheets for teaching both classical and modern cryptosystems. Buchanan [22] has also developed a similar set of worksheets for Mathematica, but mainly focusing on modern cryptosystems based on number theory.

All the CASs considered so far for teaching cryptography—Magma, Maple, Mathematica and Matlab—are closed source software. A license fee for any of those CASs can be in the hundreds of US dollars for a single student license, and up to thousands of US dollars for a department-wide license. Partly due to the costs of the above closed source CASs, there have been renewed interests since the mid-2000s to teach cryptography using open source CASs. McAndrew [66] has used the open source computer algebra systems Axiom and Maxima to teach cryptography, modelling the Axiom and Maxima computer laboratory exercises on those of May [63, 64]. Kohel [55] has taught a similar course using Sage, but with strong emphasis on mathematical foundation and algorithmic aspects of cryptography.

2.3 Sage mathematics software system

Sage [111] is an open source CAS distributed under the terms of the GNU GPL version 2 or any later version of that license [39]. The licensing terms guarantee that anyone is at liberty to copy, study, modify, improve and redistribute Sage and its source code, provided that the original terms of the license are adhered to. Started in 2005 by William Stein [110, 112], Sage is a young general purpose CAS compared to more established CASs such as Axiom, Magma, Maple, Mathematica, Matlab and Maxima. We shall not attempt a review of functionalities and features of Sage, but refer the reader to Beezer [14] and Gray [43] for such reviews.

The philosophical foundation of Sage as expressed by Joyner and Stein [51] can be summarized as applying the system of open exchange and peer review characteristic of scientific discourse to the development of mathematical software. A similar model of software development has been advocated since the early 1980s by Richard Stallman and the Free Software Foundation [109, 120], and by the open source movement [70, 89, 97] since the late 1990s. As regards the specific area of mathematical software, concerns about development being closed source and motivated by commercial interests have been voiced as early as 1995 or even earlier by Neubüser [74], who was the creator of GAP [40] for computational group theory. Neubüser's concerns are emphasized by the following quotation, which he originally wrote within the context of his discussion of the development of software tools for computational group theory, including GAP, Magma and the latter's predecessor Cayley:

> You can read Sylow's Theorem and its proof in Huppert's book in the library without even buying the book and then you can use Sylow's Theorem for the rest of your life free of charge, but ... for many computer algebra systems license fees have to be paid regularly for the total time of their use. In order to protect what you pay for, you do not get the source, but only an executable, i.e. a black box. You can press buttons and you get answers in the same way as you get the bright pictures from

> your television set but you cannot control how they were made in either case.
>
> With this situation two of the most basic rules of conduct in mathematics are violated: In mathematics information is passed on free of charge and everything is laid open for checking. Not applying these rules to computer algebra systems that are made for mathematical research ... means moving in a most undesirable direction. Most important: Can we expect somebody to believe a result of a program that he is not allowed to see? Moreover: Do we really want to charge colleagues in Moldava several years of their salary for a computer algebra system?

Similar concerns have also been expressed by 2006 Fields medalist Andrei Okounkov [71]. A prominent example in which the guiding philosophy of Sage can be seen in practice is the requirement that all changes to Sage be publicly peer reviewed.

Since its inception in 2005, Sage has been used as a computational tool in cryptography research. Albrecht [5, 6] has used the support for advanced algebra in Sage in order to mount an algebraic attack against the Courtois Toy Cipher [31]. In [59], Maitra and Sarkar employ Sage's support for computational number theory to extend the class of weak encryption exponents in RSA [98] beyond the results previously reported by Nitaj [88] and Wiener [119]. In response to the National Institute of Standards and Technology's 2007 call [87] for a proposal for a new cryptographic hash function family (SHA-3), Bertoni et al. [18] submitted the candidate Keccak family of hash functions, whose design is based in part on computational work using Sage. Further work on cryptology using Sage can be found in Albrecht and Cid [7], Albrecht et al. [8], Aner [10], Bard [12], Bernstein et al. [16, 17], Boneh et al. [20], Maitra and Sarkar [60], Velichkov et al. [116], and Weinmann [118].

Given that Sage supports functionalities for research in cryptography, how does Sage compare to other CASs in terms of support of functionalities for cryptography education? We answer this question in section 2.4, in which several CASs are compared in terms of their support for cryptography specific functionalities.

2.4 CAS functionalities for cryptography education

To support cryptography education, a CAS needs to support all or a subset of the following functionalities (this list is adapted from [66]):

- Arithmetic with arbitrary precision integers.
- String and character manipulation, including functionalities for determining ASCII codes.
- Support for common alphabets: binary, octal and hexadecimal number systems; the radix-64 alphabet; and capital letters of the English alphabet.
- Topics from elementary number theory: modular arithmetic, primitive roots, primality testing, prime number generation, integer factorization, discrete logarithms.

- Linear algebra.
- Support for knapsack and subset sum problems, including solving super-increasing sequences.
- Computation over finite fields, including manipulation of matrices with entries over finite fields.
- Elliptic curves and their arithmetic over finite fields.

We now compare six general purpose CASs with respect to the above cryptographic functionalities. Our choice of CASs are:

- FriCAS 1.0.3, released on 24 June 2008.
- Maple 12, released on May 2008.
- Mathematica 6.0, released in 2007.
- Matlab 7.2.0.283 (R2006a), released on 27 January 2006.
- Maxima 5.19.1, released on 23 August 2009.
- Sage 3.4, released on 11 March 2009.

For the purpose of comparison, we chose Sage 3.4 instead of the latest version of Sage. The primary reason is that version 3.4 is the last stable release that does not contain any of our patches to enhance the functionalities of Sage for teaching cryptography. FriCAS can be used as an optional package of Sage. As of this writing, FriCAS 1.0.3 is the latest version that is known to compile and install successfully as an optional package of the latest stable release of Sage, i.e. Sage version 4.2.1. Maxima is a standard package of Sage and Maxima 5.19.1 is the latest version that is distributed with Sage 4.2.1. As of this writing Maple 12, Mathematica 6.0 and Matlab R2006a are the latest versions installed on the machine `sage.math` [101]. This is the primary machine on which the development of our enhancement patches took place and on which the above six CASs have been successfully installed.

Tables 2.1 to 2.5 compare the level of support in FriCAS, Maple, Mathematica, Matlab, Maxima and Sage of functionalities for teaching cryptography. Table 2.1 compares support with respect to classical cryptosystems, while functionalities essential to modern number theoretic cryptosystems are compared in Table 2.2. In Table 2.3, we compare the support for various hashing functions and digital signature schemes, whereas a comparison of support for knapsack-based cryptosystems is given in Table 2.4. Finally, Table 2.5 compares support for the Data Encryption Standard (DES), the Advanced Encryption Standard (AES) and various building blocks required for these two symmetric-key cryptosystems.

A note of explanation is in order for Tables 2.1 to 2.5. The left-most column of each table lists various cryptographic functionalities that one might encounter in a cryptography course such as Barr [13], Bishop [19], Hoffstein et al. [48], Klima et al. [52], McAndrew [66], Stinson [113], and Trappe and Washington [114]. The remaining columns give an indication of how the listed cryptographic functionalities are supported in the six CASs. A functionality may be supported by a CAS, provided by third-party code, or a literature search does not reveal whether or not

the functionality is supported. For example, suppose we consider the bottom row of Table 2.1. The intersection of that row with the column for FriCAS has the acronym "AM". This indicates that the Vigenère cipher is not a functionality supported by FriCAS, but Alasdair McAndrew has provided custom written FriCAS or Axiom code to support the Vigenère cipher. Where the intersection of a row with a column contains the tick mark "✓", this indicates that the CAS considered by the column in question supports the cryptographic functionality considered by the corresponding row. Again using Table 2.1 as an example, we see that Sage 3.4 has support for the Vigenère cipher. The other acronyms are explained as follows:

- AM — code written by Alasdair McAndrew to support computer laboratory sessions of the cryptography course reported in [66].
- KSS — code by Klima et al. [52].
- MM — code by Mike May [64] available from Saint Louis University [63].
- TW — code by Trappe and Washington [114] available at their book's website [115].

If the intersection of a row with a column contains neither an acronym nor a tick mark, this indicates a lack of literature reporting on support for the corresponding cryptographic functionality in the CAS under consideration. For example, the cryptography courses [52, 64, 66, 114] do not provide any software support for the transposition cipher for FriCAS, Maple, Mathematica, Matlab or Maxima.

From Tables 2.1 to 2.5, a number of factors stand out with regards to CAS support for cryptography education. Of the six CASs surveyed, only Sage has support (either built-in or through third-party libraries distributed as standard Sage packages) for the majority of cryptographic functionalities required by undergraduate cryptography courses such as [52, 64, 66, 114]. FriCAS, Maple, Mathematica, Matlab and Maxima rely heavily on third-party custom written code to support a course in cryptography. In some cases where built-in support is lacking, there are multiple implementations of the same functionalities, as shown in Tables 2.1, 2.3, and 2.5. FriCAS and Maxima are unique in that these are the only two general purpose CASs for which a literature search has revealed third-party written code to support topics in cryptosystems based on knapsack and the subset sum problems.

To the best of our knowledge, Sage 3.4 has better support in terms of functionalities for cryptography education than either FriCAS, Maple, Mathematica, Matlab or Maxima. Bear in mind that this comparison does not take into account special purpose software such as those by Bishop [19], Chong et al. [26], Esslinger et al. [37], Patterson [92], or Spillman [108]. Extending our survey to include special purpose software tools for cryptography education would go beyond the scope of our investigation. Most of the functionalities in the Sage library, up to and including Sage version 3.4, for cryptography education are due to David R. Kohel, who is currently Professeur des universités within the Institut de Mathématiques de Luminy at the Université de la Mediterranée, France. Support for the full AES and a general framework for constructing S-boxes are due to Martin Albrecht, who is currently a PhD candidate within the Information Security Group, Royal Holloway, at the University of London, UK. The number theoretic functionalities, finite fields and elliptic curves implementations in Sage are due to the number theory development group within

the Sage Development Team, who build upon existing packages such as Givaro [42], MPIR [44], NTL [107], and Pari/GP [29].

Python is the main programming language for implementing and extending Sage. As such, the MD5 and SHA family of hash functions that are part of the Python standard library are distributed with each release of Sage. Sage 3.4 also includes the Python Cryptography Toolkit (PyCrypto) [56] as part of its repository of standard packages. PyCrypto implements various cryptographic functionalities including hash functions and public-key cryptographic algorithms. The variety of cryptographic algorithms in PyCrypto is much richer than the `hashlib` module in the Python standard library. Both `hashlib` and PyCrypto are designed to provide cryptographic services and the latter is not suitable as a software tool to support a first course in cryptography. Hence, there is a need to provide wrapper code around these two modules so that their functionalities could be used in cryptography education.

functionality	FriCAS	Maple	Mathematica	Matlab	Maxima	Sage
affine cipher		KSS, TW	TW	KSS, TW		
Caesar cipher	AM	KSS, MM, TW	TW	KSS, TW	AM	✓
Hill cipher	AM	KSS, MM, TW	TW	KSS, TW	AM	✓
shift cipher		KSS, TW	TW	KSS, TW		✓
substitution cipher						✓
transposition cipher						✓
Vigenère cipher	AM	KSS	TW	KSS, TW	AM	✓

Table 2.1: Classical cryptosystems in FriCAS, Maple, Mathematica, Matlab, Maxima and Sage.

functionality	FriCAS	Maple	Mathematica	Matlab	Maxima	Sage
Euler phi function	✓	✓	✓	TW	✓	✓
extended GCD	✓	✓	✓	✓	AM	✓
GCD	✓	✓	✓	✓	✓	✓
integer factorization	✓	✓	✓	✓	✓	✓
inverse modular arithmetic	✓	✓	✓		✓	✓
modular arithmetic	✓	✓	✓	✓	✓	✓
modular exponentiation	✓	✓	✓	TW	✓	✓
n-th prime number	AM	✓	✓	✓	AM	✓
next prime number	✓	✓	✓		✓	✓
previous prime number	✓	✓			✓	✓
primality testing	✓	✓	✓	TW	✓	✓

Table 2.2: Number theoretic functionalities in FriCAS, Maple, Mathematica, Matlab, Maxima and Sage.

functionality	FriCAS	Maple	Mathematica	Matlab	Maxima	Sage
DSS						✓
ElGamal signature scheme	AM				AM	
MD5						✓
Rabin signature scheme	AM				AM	
RSA signature scheme	AM	KSS, MM, TW	TW	KSS, TW	AM	
SHA family of hash functions						✓

Table 2.3: Hashing and digital signatures in FriCAS, Maple, Mathematica, Matlab, Maxima and Sage.

functionality	FriCAS	Maple	Mathematica	Matlab	Maxima	Sage
Merkle-Hellman additive knapsack system	AM				AM	
Merkle-Hellman multiplicative knapsack system	AM				AM	
subset sum problem	AM				AM	
super-increasing sequences	AM				AM	

Table 2.4: Knapsack cryptosystems in FriCAS, Maple, Mathematica, Matlab, Maxima and Sage.

functionality	FriCAS	Maple	Mathematica	Matlab	Maxima	Sage
AES		KSS, MM				✓
DES	AM	MM			AM	✓
elliptic curves		KSS	TW	KSS, TW		✓
finite fields	✓	✓	✓		✓	✓
S-box						✓

Table 2.5: Support for AES, DES, finite fields and elliptic curves in FriCAS, Maple, Mathematica, Matlab, Maxima and Sage.

2.5 The RSA algorithm in Sage

The six CASs surveyed in section 2.4 do not have built-in support for number theoretic cryptosystems such as RSA [98], Rabin [96] or ElGamal [36]. Sage supports various public-key cryptographic functionalities as implemented in PyCrypto, but this third-party library is not suitable as a cryptography educational tool. However, the CASs surveyed have substantial built-in support for elementary number theory so that one could easily work through the encryption and decryption procedures of the RSA, Rabin or ElGamal cryptosystems using the programming languages of these CASs. We now illustrate this using Sage for the case of the RSA encryption/decryption algorithm as presented in Algorithm 2.1.

1 Choose two primes p and q and let $n = pq$.
2 Let $e \in \mathbf{Z}$ be positive such that $\gcd\big(e, \varphi(n)\big) = 1$.
3 Let $d \in \mathbf{Z}$ be the multiplicative inverse of e modulo $\varphi(n)$.
4 Our public key is the pair (n, e) and our private key is the pair (n, d).
5 For any non-zero integer $m < n$, encrypt m using $c \equiv m^e \pmod{n}$.
6 Decrypt c using $m \equiv c^d \pmod{n}$.

Algorithm 2.1: The RSA algorithm for encryption and decryption.

As per Algorithm 2.1, we first randomly choose two positive integers and test that they are indeed primes:

```
sage: p = next_prime(randint(10^20, 10^25)); p
4685815961339311313770679
sage: q = next_prime(randint(10^20, 10^25)); q
8213166332425198564484821
sage: is_prime(p); is_prime(q)
True
True
sage: n = p * q; n
38485385893612647530529565399136160386558570363459
```

Next, we choose an integer $0 < e < \varphi(n)$ that is coprime to $\varphi(n)$. Then we compute the multiplicative inverse of e modulo $\varphi(n)$:

```
sage: e = ZZ.random_element(euler_phi(n))
sage: while gcd(e, euler_phi(n)) != 1:
....:     e = ZZ.random_element(euler_phi(n))
....:
sage: e; e < n
12036041725135809493242715057143070093942766266573
True
sage: d = inverse_mod(e, euler_phi(n)); d
14486861768954059444450932867743887374807742208797
sage: mod(d*e, euler_phi(n))
1
```

Our RSA public key is hence the pair of numbers

$$n = 38485385893612647530529565399136160386558570363459$$
$$e = 12036041725135809493242715057143070093942766266573$$

and our private key is

$$n = 384853858936126475305295653991361603865585703634 59$$
$$d = 144868617689540594444509328677438873748077422087 97.$$

Finally, we encrypt a message, decrypt the result, and verify that the message received is indeed the original message:

```
sage: m = ZZ.random_element(n); m
213499191275630924211831021443487806502165659015 22
sage: m < n
True
sage: ciphertext = power_mod(m, e, n); ciphertext
264196095179593849969166913340890664696722761192 8
sage: plaintext = power_mod(ciphertext, d, n); plaintext
213499191275630924211831021443487806502165659015 22
sage: plaintext == m
True
```

2.6 Extending Sage's cryptographic functionalities

In this section, we outline our software implementation in Sage that constitutes the body of the thesis. A review of Tables 2.1 to 2.5 suggests that one could extend the cryptographic functionalities of Sage by filling in those functionalities that are missing. Such missing functionalities include built-in support for:

- the affine and shift cryptosystems, and their cryptanalysis
- number theoretic cryptosystems such as Rabin and ElGamal
- digital signature schemes: ElGamal, Rabin, RSA
- knapsack cryptosystems and solving various classes of subset sum problems
- simplified variants of DES
- simplified variants of AES.

Due to limitation of time, we have chosen to implement the following features in the Sage standard library:

1. the shift cryptosystem and its cryptanalysis
2. the affine cryptosystem and its cryptanalysis
3. solving the subset sum problem in the particular case of super-increasing sequences
4. a simplified variant of DES
5. a simplified variant of AES.

As indicated by Table 2.1, the Caesar and shift cryptosystems have built-in support as of Sage 3.4. However, the functionalities of these two cryptosystems are simulated using the more general substitution cryptosystem. For purposes of cryptography pedagogy, we believe that functionalities of the Caesar and shift cryptosystems as well as their cryptanalysis need to be isolated in a separate Python class within the Sage standard library. Chapter 3 contains the specification of the shift cryptosystem, from which one can derive the Caesar cryptosystem as a special case; the reference manual of our Sage implementation is contained in Appendix A. The specification of the affine cryptosystem together with its cryptanalysis is presented in Chapter 4; the reference manual of our Sage implementation of this cryptosystem is contained in Appendix B. Chapters 5 and 6 present specifications of simplified variants of DES and AES, respectively, and the corresponding reference manual of our Sage implementation are given in Appendices C and D. Finally, Appendix E contains the reference manual of our implementation of an algorithm (see Proposition 6.5, pp.354–355 in [48]) for solving the subset sum problem in the specific case of super-increasing sequences. With this implementation, we hope to lay a foundation for future work on implementing various knapsack cryptosystems that use the subset sum problem over super-increasing sequences in their encryption and decryption processes.

Chapter 3

The Shift Cryptosystem

In the second century AD, the Roman historian Suetonius wrote a treatise called *Lives of the Caesars LVI* that described a cipher used by Julius Caesar (100 BC — 44 BC). The cipher is known as the Caesar shift cipher or simply the Caesar cipher. It works by moving a plaintext element by three positions along an alphabet.

This chapter describes a generalization of the Caesar cipher called the shift cryptosystem, which allows for moving a plaintext element any number of positions along an alphabet. We begin in section 3.1 with some concepts from number theory that lay the mathematical foundation of the shift cryptosystem. Section 3.2 defines the plaintext and ciphertext alphabets of the shift cryptosystem together with mathematical techniques for manipulating alphabetic elements. In section 3.3, we tie together the concepts introduced in sections 3.1 and 3.2 to define the encryption and decryption functions of the shift cryptosystem. Section 3.4 presents a number of techniques for breaking the shift cryptosystem. Ideas discussed in this chapter regarding the shift cryptosystem have been implemented as part of the Sage [111] standard library. The source code of our implementation is available with the latest stable release of Sage, which as of this writing is Sage version 4.2.1. The reference manual of our implementation is contained in Appendix A. In section 3.5, we provide numerous examples illustrating functionalities of our implementation.

3.1 Congruence and congruence classes

We begin with some results from number theory. Concepts presented in this section shall then be used in section 3.2 to define the plaintext and ciphertext alphabets of the shift cryptosystem. Our discussion in this section touches upon the theory of congruences as contained in texts such as Hungerford [49], Shoup [106] and Yan [121]. Denote by $\mathbf{Z}$ the set of all integers.

Definition 3.1. ***Divisibility.*** *Let $a, n \in \mathbf{Z}$. We say that n* divides *a if there exists some $k \in \mathbf{Z}$ such that $a = kn$. Where n divides a, we denote this relationship as $n \mid a$. Otherwise, n does not divide a and we write $n \nmid a$.*

One can show from the definition of divisibility that $0 \mid a$ if and only if $a = 0$. Furthermore, we have $n \mid a$ if and only if $-n \mid a$, which holds if and only if $n \mid -a$. In many cases, it suffices to consider $n \geq 0$.

Definition 3.2. ***Congruence.*** *Let $a, b, n \in \mathbf{Z}$ such that $n > 0$. We say that a is* congruent *to b modulo n if $n \mid (a - b)$. Where a is congruent to b modulo n, we denote this relationship as $a \equiv b \pmod{n}$.*

The relation of equality "=" on $\mathbf{Z}$ is an example of an equivalence relation (see Chapter 6 of Rosen [99] for an introduction to relations and equivalence relations). That is, if a, b, c are integers then we have the following properties with respect to equality:

1. reflexivity: $a = a$
2. symmetry: if $a = b$ then $b = a$
3. transitivity: if $a = b$ and $b = c$ then $a = c$

Congruence also shares the above three properties.

Theorem 3.3. *Congruence on $\mathbf{Z}$ is an equivalence relation.*

Proof. To show that congruence is an equivalence relation, we need to show that "$\equiv$" is reflexive, symmetric and transitive. Let $a, b, c, n \in \mathbf{Z}$ such that $n > 0$. For reflexivity, note that if $a \equiv a \pmod{n}$ then $a - a = 0$. But $n \mid (a - a)$ because $a - a = 0 = kn$ for $k = 0$.

To show symmetry, note that if $a \equiv b \pmod{n}$ then $a - b = kn$ for some integer k. Furthermore, $-(a - b) = -kn$ is equivalent to $b - a = (-k)n$ and so we have $b \equiv a \pmod{n}$ by definition of congruence.

Finally, to show transitivity, suppose $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$. Then $a - b = k_1 n$ and $b - c = k_2 n$ for some $k_1, k_2 \in \mathbf{Z}$. Add these two equations together to get $(a - b) + (b - c) = k_1 n + k_2 n$, which simplifies to $a - c = (k_1 + k_2)n$ and therefore $a \equiv c \pmod{n}$ by definition of congruence. □

For any fixed integers a and $n > 0$, there are infinitely many integers b such that a is congruent to b modulo n. To see why this is the case, note that if

$$a \equiv b \pmod{n} \tag{3.1}$$

then by Definitions 3.1 and 3.2 we have $a - b = kn$ for some $k \in \mathbf{Z}$. Solving for b produces $b = a - kn$ and substitute into the congruence (3.1) to get $a \equiv a - kn \pmod{n}$, which clearly holds for any integer values of k. This situation is captured in the following definition.

Definition 3.4. ***Congruence class.*** *Let $a, n \in \mathbf{Z}$ such that $n > 0$. The* congruence class *of a modulo n, denoted $[a]_n$, is the set of all integers congruent to a modulo n. In symbols, we have*

$$[a]_n = \{b \in \mathbf{Z} \mid a \equiv b \pmod{n}\}.$$

Definitions 3.2 and 3.4, together with Theorem 3.3 and the division algorithm (see Theorem 1.2.2, p.23 in Yan [121]), can be used to show that for any integer $n > 0$ there are n distinct congruence classes. This result relies on the idea that if two integers a and c are congruent to each other modulo n, we can equivalently treat their respective congruence classes $[a]_n$ and $[c]_n$ as being equal to each other.

Theorem 3.5. *Let $a, c, n \in \mathbf{Z}$ such that $n > 0$. Then $a \equiv c \pmod{n}$ if and only if $[a]_n = [c]_n$.*

Proof. First, we need to show that $[a]_n \subseteq [c]_n$ and $[c]_n \subseteq [a]_n$. Equality then follows by definition of equality between sets. Suppose $a \equiv c \pmod{n}$ and let $b \in [a]_n$. By definition of congruence class, we have $b \equiv a \pmod{n}$. Since $b \equiv a \pmod{n}$ and $a \equiv c \pmod{n}$, by transitivity of congruence we have $b \equiv c \pmod{n}$. Hence $b \in [c]_n$ by definition of congruence class and therefore $[a] \subseteq [c]_n$.

Assume now that $a \equiv c \pmod{n}$ and let $b \in [c]_n$. A similar argument to that in the last paragraph shows that $[c]_n \subseteq [a]_n$. Since $[a]_n \subseteq [c]_n$ and $[c]_n \subseteq [a]_n$, it follows from the definition of equality between two sets that $[a]_n = [c]_n$.

Finally, suppose that $[a]_n = [c]_n$. We need to show that $a \equiv c \pmod{n}$. Any $b \in [c]_n$ is also $b \in [a]_n$. Hence $b \in [a]_n$ implies that $b \equiv a \pmod{n}$ by definition of congruence class, or equivalently $a \equiv b \pmod{n}$ by symmetry of congruence. Similarly $b \in [c]_n$ gives $b \equiv c \pmod{n}$. Since $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$, by transitivity of congruence we have $a \equiv c \pmod{n}$. □

An immediate consequence of Theorem 3.5 is the following result.

Corollary 3.6. *Two congruence classes modulo n are either disjoint or identical.*

Proof. Consider two congruence classes $[a]_n$ and $[c]_n$ modulo n. If $[a]_n$ and $[c]_n$ are disjoint then we are done. Otherwise, suppose $[a]_n \cap [c]_n \neq \emptyset$ and let $b \in [a]_n \cap [c]_n$. We have $b \in [a]_n$ implies $b \equiv a \pmod{n}$ by definition of congruence class, and hence $a \equiv b \pmod{n}$ by symmetry of congruence. Furthermore $b \in [c]_n$ implies that $b \equiv c \pmod{n}$. Since $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$, we have $a \equiv c \pmod{n}$ by transitivity of congruence. By Theorem 3.5, $a \equiv c \pmod{n}$ is equivalent to $[a]_n = [c]_n$. □

The congruence relation modulo n partitions the set of integers into n distinct congruence classes. This result is proved in Theorem 3.7.

Theorem 3.7. *There are exactly n distinct congruence classes modulo n.*

Proof. Let $a \in \mathbf{Z}$ and let r be the remainder when a is divided by n. By the division algorithm (see p.23 in [121]), $a = nq + r$ where $0 \leq r < n$ and $q \in \mathbf{Z}$. Then $a - r = nq$ implies $a \equiv r \pmod{n}$ and therefore by Theorem 3.5, $a \equiv r \pmod{n}$ is equivalent to $[a]_n = [r]_n$.

If $[a]_n$ is any congruence class modulo n, we now show that $[a]_n$ is one of the congruence classes $[0]_n, [1]_n, [2]_n, \ldots, [n-1]_n$. Note that from the last paragraph we have $[a]_n = [r]_n$ where r is the remainder when a is divided by n and $0 \leq r < n$. Hence $[a]_n \in \{[0]_n, [1]_n, [2]_n, \ldots, [n-1]_n\}$. For the case of pair-wise distinctness, let $s, t \in \mathbf{Z}$ such that $0 \leq s < t < n$. Then the positive integer $t - s < n$ implies that $n \nmid (t - s)$. Hence we have $t \not\equiv s \pmod{n}$ and therefore $[t]_n \neq [s]_n$ by Theorem 3.5. □

The n distinct congruence classes of Theorem 3.7 can be summarized in the following definition. We also refer to the set $\mathbf{Z}/n\mathbf{Z}$ in Definition 3.8 as the set of integers modulo n. From hereon, we shall use the notation $a \in \mathbf{Z}/n\mathbf{Z}$ instead of $[a]_n \in \mathbf{Z}/n\mathbf{Z}$ since one can define the set of integers modulo n as $\mathbf{Z}/n\mathbf{Z} = \{0, 1, 2, \ldots, n-1\}$.

Definition 3.8. ***Integers modulo n.*** *The set of all congruence classes modulo n is denoted $\mathbf{Z}/n\mathbf{Z}$.*

3.2 Plaintext and ciphertext alphabets

By now, we have covered the necessary number theoretic concepts to allow for a discussion of the plaintext and ciphertext alphabets of the shift cryptosystem. Let $\mathcal{A} = \{a_0, a_1, a_2, \ldots, a_{n-1}\}$ be a non-empty alphabet consisting of n elements. Define a mapping

$$f : \mathcal{A} \longrightarrow \mathbf{Z}/n\mathbf{Z} \tag{3.2}$$

given by $f(a_i) = i$, which uniquely assigns each alphabetic element $a_i \in \mathcal{A}$ to a congruence class $i \in \mathbf{Z}/n\mathbf{Z}$. It is thus clear that the map (3.2) is bijective. Both the plaintext and its corresponding ciphertext are encoded using elements of $\mathcal{A}$, so that $\mathcal{A}$ is considered as both the plaintext space and ciphertext space. If $\mathcal{A}$ denotes the capital letters of the English alphabet, then Table 3.1 shows the mapping of each letter to its integer equivalent.

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25

Table 3.1: Assigning capital letters of the English alphabet to numbers.

3.3 Encryption and decryption functions

The shift cryptosystem is a symmetric-key cryptosystem in which each secret key k is an element of $\mathbf{Z}/n\mathbf{Z}$. By Theorem 3.7, it is clear that the key space $\mathbf{Z}/n\mathbf{Z}$ consists of n possible keys.

Let $\mathbf{P} = (p_0, p_1, p_2, \ldots, p_{m-1})$ be a non-empty plaintext consisting of m elements, each of which is encoded as an element of $\mathcal{A}$ and by (3.2) we have $p_i \in \mathbf{Z}/n\mathbf{Z}$. For each plaintext element p, the encryption function $\mathcal{E} : \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z} \longrightarrow \mathbf{Z}/n\mathbf{Z}$ uses the secret key k to produce a corresponding ciphertext c:

$$\mathcal{E}(k, p) = p + k \pmod{n}. \tag{3.3}$$

One can think of $\mathcal{E}$ as shifting p along $\mathcal{A}$ by k positions with wrap around. Applying the encryption function $\mathcal{E}$ to each p_i of $\mathbf{P}$ results in the ciphertext $\mathbf{C} = (c_0, c_1, c_2, \ldots, c_{m-1})$.

We can recover the plaintext as follows. Given a ciphertext element c and a secret key k, the decryption function $\mathcal{D} : \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z} \longrightarrow \mathbf{Z}/n\mathbf{Z}$ can be described similarly to $\mathcal{E}$:

$$\mathcal{D}(k, c) = c - k \pmod{n}. \tag{3.4}$$

This decryption process can be interpreted as moving c along $\mathcal{A}$ by $n - k$ positions with wrap around. The bijection (3.2) is then used to convert elements of the plaintext and ciphertext to alphabetic elements. We have implemented the shift cryptosystem in the class

`sage.crypto.classical.ShiftCryptosystem`

of the Sage standard library. Refer to Appendix A for the reference manual of our implementation.

Notice that if an integer k is in the key space $\mathbf{Z}/n\mathbf{Z}$, then its corresponding inverse key is $-k \pmod{n}$. This enables one to express both the encryption and decryption processes as a general function of the form

$$\mathcal{F}(K, m) = m + K \pmod{n}. \tag{3.5}$$

If K is a secret key and m is a plaintext element, then $\mathcal{F}(K, m)$ defines the encryption function $\mathcal{E}$ in (3.3). On the other hand, where K is the inverse key of a secret key and m is a ciphertext element, then $\mathcal{F}(K, m)$ defines the decryption function $\mathcal{D}$ in (3.4).

3.4 Cryptanalysis

This section discusses a number of techniques that can be used to break the shift cryptosystem. We begin with the technique of brute-force, otherwise known as exhaustive key search.

3.4.1 Exhaustive key search

The shift cryptosystem is vulnerable to a brute-force attack. The size of the key space is precisely the number of elements in the alphabet under consideration. Since our key space is $\mathbf{Z}/n\mathbf{Z}$, then by Theorem 3.7 we need only perform at most n searches in order to recover a secret key. Given a non-empty ciphertext $\mathbf{C}$, we can search the key space $\mathbf{Z}/n\mathbf{Z}$, decrypting $\mathbf{C}$ using each candidate key $k \in \mathbf{Z}/n\mathbf{Z}$ to obtain n candidate decipherments $\mathbf{P}_0, \mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_{n-1}$. Next, use the map (3.2) to transform each $\mathbf{P}_i$ to elements of the alphabet $\mathcal{A}$ and observe which candidate decipherment results in something meaningful. After using (3.2) to encode $\mathbf{P}_k$ as elements of $\mathcal{A}$, if the result is plausible plaintext, then k is the secret key. The brute-force attack against the shift cryptosystem is implemented in the method

`sage.crypto.classical.ShiftCryptosystem.brute_force`

of the Sage standard library. See section A.2.1 for the reference manual of this method.

3.4.2 Monogram frequency analysis

The technique of frequency analysis uses the empirical observation that certain elements of $\mathcal{A}$ and various permutations of subsets of $\mathcal{A}$ occur more frequently than others. If $\mathbf{P}$ is a non-empty plaintext encoded using letters of the upper-case English alphabet, and $\mathbf{C}$ is the ciphertext corresponding to $\mathbf{P}$, a frequency analysis attack on $\mathbf{C}$ uses statistical properties of the English language.

Monogram frequency analysis specializes frequency analysis to single alphabetic elements or monograms. This attack performs frequency analysis on monograms (occurrences of each single letter), as opposed to frequency analysis of digrams (occurrences of letter pairs) or trigrams (occurrences of letter triples). In written English,

various letters of the English alphabet occur more frequently than others. The letter E appears more often than other vowels such as A, I, O and U. When a table of relative frequencies is compiled from various written sources such as novels, magazines, newspapers and literary works, one can observe that the probability of a letter occurring tends to stabilize around a certain value. We call this value the characteristic frequency probability (CFP) of the letter under consideration. When this probability is considered for each letter of the English alphabet, the resulting probabilities for all letters of that alphabet is referred to as the characteristic frequency probability distribution or CFP distribution. Various studies report slightly different values for the CFP of an English letter. Lewand [57] reports that E has a CFP of 0.12702, while Beker and Piper [15] report this value as 0.127. Table 3.2 shows the CFP distribution of Beker and Piper [15], whereas Table 3.3 shows the corresponding CFP distribution of Lewand [57]. The concepts of CFP and CFP distribution can also be applied to alphabets other than the English alphabet.

Letter	CFP	Letter	CFP
A	0.082	N	0.067
B	0.015	O	0.075
C	0.028	P	0.019
D	0.043	Q	0.001
E	0.127	R	0.060
F	0.022	S	0.063
G	0.020	T	0.091
H	0.061	U	0.028
I	0.070	V	0.010
J	0.002	W	0.023
K	0.008	X	0.001
L	0.040	Y	0.020
M	0.024	Z	0.001

Table 3.2: The characteristic frequency probability distribution of Beker and Piper [15].

Given a CFP distribution such as Tables 3.2 or 3.3, and a non-empty ciphertext sample $\mathbf{C}$ encoded using upper-case letters of the English alphabet, we construct a table of relative frequencies of letters in $\mathbf{C}$. We refer to this table as the frequency probability (FP) distribution corresponding to $\mathbf{C}$. Let $\mathbf{T} = (t_0, t_1, t_2, \ldots, t_{25})$ be a vector of the FP values in an FP distribution where $t_i \geq t_j$ for all $i, j \in \mathbf{Z}/n\mathbf{Z}$ such that $i \neq j$. Let $\mathbf{A} = (a_0, a_1, a_2, \ldots, a_{25})$ be such that a_i is the alphabetic element with the FP value t_i. Starting with t_0, we make a guess that a_0 is the ciphertext corresponding to E. From Table 3.1 we have the mapping E $\longmapsto$ 4 and by (3.2) we know that $a_0 \longmapsto j$ for some $j \in \mathbf{Z}/n\mathbf{Z}$. Then the (candidate) secret key is $k \equiv j - 4 \pmod{26}$. Apply Table 3.1 to encode the candidate decipherment $\mathbf{P}_k$ using alphabetic characters. If the result is not a meaningful plaintext, we choose a_1, make a guess that a_1 is the ciphertext letter corresponding to E, and repeat the above procedure to obtain a candidate decipherment. In this way, we work through each $a_i \in \mathbf{A}$ for $i = 0, 1, 2, \ldots, 25$ until we find some a_j that results in a meaningful plaintext. The technique of monogram frequency analysis can be extended to any alphabet that has a corresponding CFP distribution.

Letter	CFP	Letter	CFP
A	0.08167	N	0.06749
B	0.01492	O	0.07507
C	0.02782	P	0.01929
D	0.04253	Q	0.00095
E	0.12702	R	0.05987
F	0.02228	S	0.06327
G	0.02015	T	0.09056
H	0.06094	U	0.02758
I	0.06966	V	0.00978
J	0.00153	W	0.02360
K	0.00772	X	0.00150
L	0.04025	Y	0.01974
M	0.02406	Z	0.00074

Table 3.3: The characteristic frequency probability distribution of Lewand [57].

3.4.3 Ranking candidate keys

The technique of frequency analysis of monograms can be combined with a number of elementary statistical measures in order to provide a procedure for ranking each key in the key space. Let $\mathbf{C}$ be a non-empty ciphertext corresponding to some plaintext $\mathbf{P}$ and let $\mathbf{P}_k$ be a candidate decipherment of $\mathbf{C}$. In other words, $\mathbf{P}_k$ is the result of attempting to decrypt $\mathbf{C}$ using a candidate key $k \in \mathbf{Z}/n\mathbf{Z}$ which is not necessarily the same key used to encrypt $\mathbf{P}$. Denote by $F_{\mathcal{A}}(e)$ the characteristic frequency probability of $e \in \mathcal{A}$ and let $F_{\mathbf{P}_k}(e)$ be the relative frequency of e as observed in $\mathbf{P}_k$. The CFP distribution of an alphabet $\mathcal{A}$ can be considered as the expected frequency probability distribution for that alphabet. The relative frequency probability distribution of $\mathbf{P}_k$ provides a distribution of the ratio of character occurrences over message length, i.e. the length of $\mathbf{P}_k$. One can consider $F_{\mathcal{A}}(e)$ as the expected probability, while $F_{\mathbf{P}_k}(e)$ can be considered as the observed probability.

If $\mathbf{P}_k$ is of length L, then the observed frequency of $e \in \mathcal{A}$ is

$$O_{\mathbf{P}_k}(e) = F_{\mathbf{P}_k}(e) \cdot L \tag{3.6}$$

and the expected frequency of $e \in \mathcal{A}$ is

$$E_{\mathcal{A}}(e) = F_{\mathcal{A}}(e) \cdot L. \tag{3.7}$$

The squared-differences, or residual sum of squares, rank $R_{RSS}(\mathbf{P}_k)$ of $\mathbf{P}_k$ corresponding to a candidate key $k \in \mathbf{Z}/n\mathbf{Z}$ is given by

$$R_{RSS}(\mathbf{P}_k) = \sum_{e \in \mathcal{A}} \big(O_{\mathbf{P}_k}(e) - E_{\mathcal{A}}(e)\big)^2$$

where the sum is taken over all alphabetic elements. Cryptanalysis by exhaustive key search produces a candidate decipherment $\mathbf{P}_k$ for each possible key $k \in \mathbf{Z}/n\mathbf{Z}$. Given a set $D = \{\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_{k_r}\}$ of all candidate decipherments corresponding to $\mathbf{C}$, the smaller the rank $R_{RSS}(\mathbf{P}_{k_i})$, the more likely it is that k_i is the secret key. This key ranking method is based on the residual sum of squares measure.

We can also define a key ranking function based on the chi-square statistical measure. Let $O_{\mathbf{P}_k}(e)$ and $E_{\mathcal{A}}(e)$ be as in (3.6) and (3.7), respectively. The chi-square rank $R_{\chi^2}(\mathbf{P}_k)$ of $\mathbf{P}_k$ corresponding to a candidate key $k \in \mathbf{Z}/n\mathbf{Z}$ is given by

$$R_{\chi^2}(\mathbf{P}_k) = \sum_{e \in \mathcal{A}} \frac{\big(O_{\mathbf{P}_k}(e) - E_{\mathcal{A}}(e)\big)^2}{E_{\mathcal{A}}(e)}$$

where the sum is taken over all alphabetic elements. We have implemented the above two key ranking techniques for the shift cryptosystem in the methods

`sage.crypto.classical.ShiftCryptosystem.rank_by_chi_square`
`sage.crypto.classical.ShiftCryptosystem.rank_by_squared_differences`

of the Sage standard library. Refer to sections A.2.7 and A.2.8, respectively, for the reference manual of these two methods.

3.5 Example Sage usage

This section provides examples illustrating functionalities of our Sage [111] implementation of the shift cryptosystem. The reference manual of our implementation is contained in Appendix A and the source code is available with the latest stable release of Sage.

Here we provide some examples illustrating encryption and decryption over various alphabets. Here is an example over the upper-case letters of the English alphabet:

```
sage: S = ShiftCryptosystem(AlphabeticStrings()); S
Shift cryptosystem on Free alphabetic string monoid on A-Z
sage: P = S.encoding("The shift cryptosystem generalizes the Caesar cipher.")
sage: P
THESHIFTCRYPTOSYSTEMGENERALIZESTHECAESARCIPHER
sage: K = 7
sage: C = S.enciphering(K, P); C
AOLZOPMAJYFWAVZFZALTNLULYHSPGLZAOLJHLZHYJPWOLY
sage: S.deciphering(K, C)
THESHIFTCRYPTOSYSTEMGENERALIZESTHECAESARCIPHER
sage: S.deciphering(K, C) == P
True
```

The previous example can also be worked through as follows using functional notation:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("The shift cryptosystem generalizes the Caesar cipher.")
sage: K = 7
sage: E = S(K); E
Shift cipher on Free alphabetic string monoid on A-Z
sage: C = E(P); C
AOLZOPMAJYFWAVZFZALTNLULYHSPGLZAOLJHLZHYJPWOLY
sage: D = S(S.inverse_key(K)); D
Shift cipher on Free alphabetic string monoid on A-Z
sage: D(C) == P
True
sage: D(C) == P == D(E(P))
True
```

Here is an example over the hexadecimal number system:

```
sage: S = ShiftCryptosystem(HexadecimalStrings()); S
Shift cryptosystem on Free hexadecimal string monoid
sage: P = S.encoding("Using hexadecimal numbers."); P
5573696e67206865786164656369 6d616c206e756d626572732e
sage: K = 5
sage: C = S.enciphering(K, P); C
aac8beb3bc75bdbacdb6b9bab8beb2b6b175b3cab2b7bac7c873
sage: S.deciphering(K, C)
5573696e67206865786164656369 6d616c206e756d626572732e
sage: S.deciphering(K, C) == P
True
```

An example over the binary number system:

```
sage: S = ShiftCryptosystem(BinaryStrings()); S
Shift cryptosystem on Free binary string monoid
sage: P = S.encoding("insecure"); P
011010010110111001110011011001010110001101110101011100100110010 1
sage: K = 1
sage: C = S.enciphering(K, P); C
100101101001000110001100100110101001110010001010100011011001101 0
sage: S.deciphering(K, C)
011010010110111001110011011001010110001101110101011100100110010 1
sage: S.deciphering(K, C) == P
True
```

A shift cryptosystem with key $k = 3$ is commonly referred to as the Caesar cipher. Create a Caesar cipher over the upper-case letters of the English alphabet:

```
sage: caesar = ShiftCryptosystem(AlphabeticStrings())
sage: K = 3
sage: P = caesar.encoding("abcdefghijklmnopqrstuvwxyz"); P
ABCDEFGHIJKLMNOPQRSTUVWXYZ
sage: C = caesar.enciphering(K, P); C
DEFGHIJKLMNOPQRSTUVWXYZABC
sage: caesar.deciphering(K, C) == P
True
```

Generate a random key for encryption and decryption:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Shift cipher with a random key.")
sage: K = S.random_key()
sage: C = S.enciphering(K, P)
sage: S.deciphering(K, C) == P
True
```

Decrypting with the key `K` is equivalent to encrypting with its corresponding inverse key:

```
sage: S.enciphering(S.inverse_key(K), C) == P
True
```

Cryptanalyze over the capital letters of the English alphabet using all possible keys:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("The shift cryptosystem generalizes the Caesar cipher.")
sage: K = 7
sage: C = S.enciphering(K, P)
sage: Dict = S.brute_force(C)
sage: for k in xrange(len(Dict)):
...       if Dict[k] == P:
...           print "key =", k
...
key = 7
```

We can perform an exhaustive key search only, without using any ranking functions:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Shifting using modular arithmetic.")
sage: K = 8
sage: C = S.enciphering(K, P)
sage: pdict = S.brute_force(C)
sage: sorted(pdict.items())
<BLANKLINE>
[(0, APQNBQVOCAQVOUWLCTIZIZQBPUMBQK),
(1, ZOPMAPUNBZPUNTVKBSHYHYPAOTLAPJ),
(2, YNOLZOTMAYOTMSUJARGXGXOZNSKZOI),
(3, XMNKYNSLZXNSLRTIZQFWFWNYMRJYNH),
(4, WLMJXMRKYWMRKQSHYPEVEVMXLQIXMG),
(5, VKLIWLQJXVLQJPRGXODUDULWKPHWLF),
(6, UJKHVKPIWUKPIOQFWNCTCTKVJOGVKE),
(7, TIJGUJOHVTJOHNPEVMBSBSJUINFUJD),
(8, SHIFTINGUSINGMODULARARITHMETIC),
(9, RGHESHMFTRHMFLNCTKZQZQHSGLDSHB),
(10, QFGDRGLESQGLEKMBSJYPYPGRFKCRGA),
(11, PEFCQFKDRPFKDJLARIXOXOFQEJBQFZ),
(12, ODEBPEJCQOEJCIKZQHWNWNEPDIAPEY),
(13, NCDAODIBPNDIBHJYPGVMVMDOCHZODX),
(14, MBCZNCHAOMCHAGIXOFULULCNBGYNCW),
(15, LABYMBGZNLBGZFHWNETKTKBMAFXMBV),
(16, KZAXLAFYMKAFYEGVMDSJSJALZEWLAU),
(17, JYZWKZEXLJZEXDFULCRIRIZKYDVKZT),
(18, IXYVJYDWKIYDWCETKBQHQHYJXCUJYS),
(19, HWXUIXCVJHXCVBDSJAPGPGXIWBTIXR),
(20, GVWTHWBUIGWBUACRIZOFOFWHVASHWQ),
(21, FUVSGVATHFVATZBQHYNENEVGUZRGVP),
(22, ETURFUZSGEUZSYAPGXMDMDUFTYQFUO),
(23, DSTQETYRFDTYRXZOFWLCLCTESXPETN),
(24, CRSPDSXQECSXQWYNEVKBKBSDRWODSM),
(25, BQROCRWPDBRWPVXMDUJAJARCQVNCRL)]
```

Combine exhaustive key search with the chi-square and squared-differences ranking functions. With sufficiently long ciphertext, both of these ranking functions give the same rank to the secret key:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Shifting using modular arithmetic.")
sage: K = 8
sage: C = S.enciphering(K, P)
sage: S.brute_force(C, ranking="chisquare")
<BLANKLINE>
[(8, SHIFTINGUSINGMODULARARITHMETIC),
(14, MBCZNCHAOMCHAGIXOFULULCNBGYNCW),
(20, GVWTHWBUIGWBUACRIZOFOFWHVASHWQ),
(13, NCDAODIBPNDIBHJYPGVMVMDOCHZODX),
(1, ZOPMAPUNBZPUNTVKBSHYHYPAOTLAPJ),
(23, DSTQETYRFDTYRXZOFWLCLCTESXPETN),
```

```
(10, QFGDRGLESQGLEKMBSJYPYPGRFKCRGA),
(6, UJKHVKPIWUKPIOQFWNCTCTKVJOGVKE),
(22, ETURFUZSGEUZSYAPGXMDMDUFTYQFUO),
(15, LABYMBGZNLBGZFHWNETKTKBMAFXMBV),
(12, ODEBPEJCQOEJCIKZQHWNWNEPDIAPEY),
(21, FUVSGVATHFVATZBQHYNENEVGUZRGVP),
(16, KZAXLAFYMKAFYEGVMDSJSJALZEWLAU),
(25, BQROCRWPDBRWPVXMDUJAJARCQVNCRL),
(9, RGHESHMFTRHMFLNCTKZQZQHSGLDSHB),
(24, CRSPDSXQECSXQWYNEVKBKBSDRWODSM),
(3, XMNKYNSLZXNSLRTIZQFWFWNYMRJYNH),
(5, VKLIWLQJXVLQJPRGXODUDULWKPHWLF),
(7, TIJGUJOHVTJOHNPEVMBSBSJUINFUJD),
(2, YNOLZOTMAYOTMSUJARGXGXOZNSKZOI),
(18, IXYVJYDWKIYDWCETKBQHQHYJXCUJYS),
(4, WLMJXMRKYWMRKQSHYPEVEVMXLQIXMG),
(11, PEFCQFKDRPFKDJLARIXOXOFQEJBQFZ),
(19, HWXUIXCVJHXCVBDSJAPGPGXIWBTIXR),
(0, APQNBQVOCAQVOUWLCTIZIZQBPUMBQK),
(17, JYZWKZEXLJZEXDFULCRIRIZKYDVKZT)]
sage: S.brute_force(C, ranking="squared_differences")
<BLANKLINE>
[(8, SHIFTINGUSINGMODULARARITHMETIC),
(23, DSTQETYRFDTYRXZOFWLCLCTESXPETN),
(12, ODEBPEJCQOEJCIKZQHWNWNEPDIAPEY),
(2, YNOLZOTMAYOTMSUJARGXGXOZNSKZOI),
(9, RGHESHMFTRHMFLNCTKZQZQHSGLDSHB),
(7, TIJGUJOHVTJOHNPEVMBSBSJUINFUJD),
(21, FUVSGVATHFVATZBQHYNENEVGUZRGVP),
(22, ETURFUZSGEUZSYAPGXMDMDUFTYQFUO),
(1, ZOPMAPUNBZPUNTVKBSHYHYPAOTLAPJ),
(16, KZAXLAFYMKAFYEGVMDSJSJALZEWLAU),
(20, GVWTHWBUIGWBUACRIZOFOFWHVASHWQ),
(24, CRSPDSXQECSXQWYNEVKBKBSDRWODSM),
(14, MBCZNCHAOMCHAGIXOFULULCNBGYNCW),
(13, NCDAODIBPNDIBHJYPGVMVMDOCHZODX),
(3, XMNKYNSLZXNSLRTIZQFWFWNYMRJYNH),
(10, QFGDRGLESQGLEKMBSJYPYPGRFKCRGA),
(15, LABYMBGZNLBGZFHWNETKTKBMAFXMBV),
(6, UJKHVKPIWUKPIOQFWNCTCTKVJOGVKE),
(11, PEFCQFKDRPFKDJLARIXOXOFQEJBQFZ),
(25, BQROCRWPDBRWPVXMDUJAJARCQVNCRL),
(17, JYZWKZEXLJZEXDFULCRIRIZKYDVKZT),
(19, HWXUIXCVJHXCVBDSJAPGPGXIWBTIXR),
(4, WLMJXMRKYWMRKQSHYPEVEVMXLQIXMG),
(0, APQNBQVOCAQVOUWLCTIZIZQBPUMBQK),
(18, IXYVJYDWKIYDWCETKBQHQHYJXCUJYS),
(5, VKLIWLQJXVLQJPRGXODUDULWKPHWLF)]
```

Here is an example where the chi-square ranking function out-performs the squared-differences ranking function:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Short")
sage: K = 11
sage: C = S.enciphering(K, P)
sage: S.brute_force(C, ranking="chisquare")
<BLANKLINE>
[(11, SHORT),
 (25, ETADF),
 (10, TIPSU),
 (22, HWDGI),
 (17, MBILN),
 (1, CRYBD),
 (24, FUBEG),
 (18, LAHKM),
 (15, ODKNP),
```

```
 (23, GVCFH),
 (8, VKRUW),
 (16, NCJMO),
 (20, JYFIK),
 (0, DSZCE),
 (12, RGNQS),
 (14, PELOQ),
 (6, XMTWY),
 (13, QFMPR),
 (3, APWZB),
 (4, ZOVYA),
 (7, WLSVX),
 (21, IXEHJ),
 (9, UJQTV),
 (19, KZGJL),
 (5, YNUXZ),
 (2, BQXAC)]
sage: S.brute_force(C, ranking="squared_differences")
<BLANKLINE>
[(25, ETADF),
 (11, SHORT),
 (10, TIPSU),
 (0, DSZCE),
 (14, PELOQ),
 (21, IXEHJ),
 (22, HWDGI),
 (17, MBILN),
 (18, LAHKM),
 (15, ODKNP),
 (24, FUBEG),
 (12, RGNQS),
 (16, NCJMO),
 (4, ZOVYA),
 (1, CRYBD),
 (6, XMTWY),
 (23, GVCFH),
 (3, APWZB),
 (7, WLSVX),
 (9, UJQTV),
 (8, VKRUW),
 (2, BQXAC),
 (13, QFMPR),
 (20, JYFIK),
 (5, YNUXZ),
 (19, KZGJL)]
```

Chapter 4

The Affine Cryptosystem

Whereas the shift cryptosystem is a generalization of the Caesar cipher, the affine cryptosystem is in turn a generalization of the shift cryptosystem. The affine cryptosystem is so named since its encryption and decryption functions are affine or linear functions.

This chapter generalizes the shift cryptosystem discussed in Chapter 3. Section 4.1 continues the development of number theoretic techniques started in Chapter 3. The number theoretic techniques are then applied in section 4.2 to analyze the key space of the affine cryptosystem within an algebraic setting. In section 4.3, we tie together the concepts and techniques developed in sections 4.1 and 4.2 to define the encryption and decryption functions of the affine cryptosystem. This is followed by a brief discussion in section 4.4 of cryptanalytic techniques that can be brought to bear on the affine cryptosystem. We have implemented the affine cryptosystem discussed in this chapter as part of the Sage [111] standard library. The source code of our implementation is available with the latest stable release of Sage, which as of this writing is Sage 4.2.1. The reference manual of our implementation is contained in Appendix B. Examples illustrating functionalities of our implementation are presented in section 4.5.

4.1 Greatest common divisors

This section continues the development of the theory of divisibility and congruence began in section 3.1. Number theoretic concepts discussed in this section play a vital role in analyzing the structure of the key space of the affine cryptosystem. As discussed in section 3.1, the notion of divisibility allows one to define congruence, congruence classes, and use these concepts to prove that a shift cryptosystem whose key space is $\mathbf{Z}/n\mathbf{Z}$, for any integer $n > 0$, has exactly n possible keys. This section follows in the footsteps of section 3.1 and develops necessary concepts and techniques to understand the key space of the affine cryptosystem. Our discussion touches upon algebraic and number theoretic techniques as contained in algebra and number theory texts such as Hungerford [49], Shoup [106] and Yan [121].

We begin with a definition of greatest common divisors.

Definition 4.1. ***Greatest common divisor.*** *Let $a, b \in \mathbf{Z}$ such that at least one of them is not zero. The* greatest common divisor *of a and b, in symbols* $\gcd(a, b)$*, is the positive integer d such that*

1. $d \mid a$ *and* $d \mid b$
2. *if* $c \mid a$ *and* $c \mid b$ *then* $c \leq d$.

We can extend this definition by setting $\gcd(0,0) = 0$.

If $\gcd(a,b) = d$ for some integers a and b, we can express d as a linear combination of a and b. This result is known as Bézout's identity. Before proving Bézout's identity, we first prove an intermediate result.

Theorem 4.2. *Let* $a, b, c, m \in \mathbf{Z}$ *and consider the equation*

$$a = b + c. \tag{4.1}$$

If m *divides any two of* a, b, c *in (4.1), then* m *divides the third.*

Proof. Let $a, b, c \in \mathbf{Z}$ such that $a = b + c$ and suppose m divides any two terms in the latter equation. We distinguish three separate cases.

1. If $m \mid a$ and $m \mid b$ then by definition $a = mx$ and $b = my$ for $x, y \in \mathbf{Z}$. Substituting these into the equation and simplifying produces $mx = (my) + c$, which implies that $c = m(x - y)$ and hence $m \mid c$.

2. If $m \mid a$ and $m \mid c$ then it follows that $a = mx$ and $c = mz$ for integers x and z. Substituting these into the equation and simplifying results in $mx = mz + b$. Solving for b gives $b = m(x - z)$ and hence $m \mid b$.

3. If $m \mid b$ and $m \mid c$ then by definition we have $b = my$ and $c = mz$ for integers y, z. Substituting these into the equation and simplifying yields $a = m(y + z)$ and it follows that $m \mid a$.

In each of the above three cases, we see that whenever m divides any two terms in equation (4.1) then m also divides the third term. □

We now use Theorem 4.2 to prove Bézout's identity.

Theorem 4.3. ***Bézout's identity.*** *Let* $a, b \in \mathbf{Z}$*, one of which is non-zero. Then there exist* $x, y \in \mathbf{Z}$ *such that* $\gcd(a,b) = ax + by$.

Proof. Let S be the set all linear combinations defined as

$$S = \{au + bv \mid u, v \in \mathbf{Z}\}.$$

Let $c = am + bn$ be the smallest positive integer in S for some $m, n \in \mathbf{Z}$. We first show that c divides both a and b. Since $c > 0$ and $a \in \mathbf{Z}$, use the division algorithm to obtain $q, r \in \mathbf{Z}$ such that $a = qc + r$, where $0 \leq r < c$. Express the remainder r as $r = a - qc$ and, since $c = am + bn$, so we have $r = a - q(am + bn) = a(1 - mq) - bnq$, which implies that $r = a(1 - mq) + b(-nq)$. By hypothesis $0 \leq r < c$ and if $r \neq 0$ then $r > 0$ such that $r < c$. But then r is expressible as a linear combination in terms of a and b and hence $r \in S$ by definition of S. Here lies a contradiction because $c \in S$ is by hypothesis the smallest positive integer expressible as a linear combination of a and b. Hence $r = 0$. Since $a = qc + r$ and $r = 0$, then $a = qc$ and hence $c \mid a$. A similar argument shows that c divides b.

Next, we show that $c = \gcd(a, b)$. Let $d = \gcd(a, b)$ so that $d \mid a$ and $d \mid b$. By Theorem 4.2 we have $d \mid c$ and by Definition 3.1, $c = dz$ for some $z \in \mathbf{Z}$. Since $c > 0$ and $d > 0$ by hypothesis, then $z > 0$. If $z > 1$ then c is a common divisor of both a and b such that $c > d$, in contradiction of our hypothesis that $d = \gcd(a, b)$. Thus $z = 1$ and it follows that $c = d$. Therefore $\gcd(a, b) = ax + by$ for some $x, y \in \mathbf{Z}$. □

It should be noted that Bézout's identity is an existence result. Given $a, b \in \mathbf{Z}$, it guarantees the existence of $x, y \in \mathbf{Z}$ such that $\gcd(a, b) = ax + by$ without providing an algorithm for finding specific values of x and y. The values of x and y whose existence is guaranteed by Bézout's identity can be computed using the extended Euclidean algorithm (see section 4.2, pp.77–81 of Shoup [106]). However, the pair (x, y) is not unique. Given a pair (x, y) satisfying Bézout's identity for some fixed integers a and b, the set

$$\left\{ \left(x + \frac{kb}{\gcd(a,b)}, y - \frac{ka}{\gcd(a,b)} \right) \;\middle|\; k \in \mathbf{Z} \right\}$$

provides an infinite pairs of integers that also satisfy Bézout's identity for the chosen a and b. The following result is an immediate consequence of Bézout's identity.

Corollary 4.4. *If $a \mid bc$ and $\gcd(a, b) = 1$ then $a \mid c$.*

Proof. If $a \mid bc$ with $\gcd(a, b) = 1$, use Theorem 4.3 to obtain

$$1 = \gcd(a, b) = ax + by$$

for some $x, y \in \mathbf{Z}$. Multiply the last equation through by c to get

$$c = axc + byc.$$

Since $a \mid ac$ and $a \mid bc$, it follows that $a \mid axc$ and $a \mid byc$. Therefore $a \mid (axc + byc)$ as required. □

Theorem 4.5. *The linear congruence $ax \equiv b \pmod{n}$ has a unique solution $x \in \mathbf{Z}/n\mathbf{Z}$ for any $b \in \mathbf{Z}/n\mathbf{Z}$ if and only if $\gcd(a, n) = 1$.*

Proof. First, we need to show that if $ax \equiv b \pmod{n}$ has a unique solution for all b, then $\gcd(a, n) = 1$. Proving this statement is equivalent to showing that its contrapositive holds. Towards that end, suppose that $\gcd(a, n) > 1$ and consider the congruence

$$ax \equiv 0 \pmod{n}. \tag{4.2}$$

Clearly $x = 0$ is a solution of (4.2) as well as $x = n/\gcd(a, n)$, thus proving the contrapositive.

Now let $\gcd(a, n) = 1$. If $x_1, x_2 \in \mathbf{Z}/n\mathbf{Z}$ are two solutions of $ax \equiv b \pmod{n}$ then $ax_1 \equiv ax_2 \pmod{n}$. By Definition 3.2 we have $n \mid a(x_1 - x_2)$. Since $\gcd(a, n) = 1$ and $n \mid a(x_1 - x_2)$, then $n \mid (x_1 - x_2)$ by Corollary 4.4. Conclude by Definition 3.2 that $x_1 \equiv x_2 \pmod{n}$ and therefore the solution to $ax \equiv b \pmod{n}$ is unique. □

4.2 Multiplicative groups

Building on the number theoretic techniques of section 4.1, this section reviews a number of results relating to an algebraic object known as a group. We then

analyze the structure of a special subset of $\mathbf{Z}/n\mathbf{Z}$, known as its multiplicative group, in terms of group theoretic techniques and the Euler phi function. As we shall see in section 4.3, the multiplicative group of $\mathbf{Z}/n\mathbf{Z}$ is crucial in defining the key space of the affine cryptosystem and obtaining an explicit formula that expresses the size of that space. See Hungerford [49] for further discussion on group theory, or Shoup [106] for an interplay between algebra and number theory.

We begin with a definition of groups.

Definition 4.6. ***Group.*** *A* group *is a non-empty set G equipped with a binary operation $*$ that satisfies the following axioms:*

1. *Closure: If $a, b \in G$ then $a * b \in G$.*
2. *Associativity: $a * (b * c) = (a * b) * c$ for all $a, b, c \in G$.*
3. *Identity element: There is an element $e \in G$ such that $a * e = a = e * a$ for all $a \in G$.*
4. *Inverse: For each $a \in G$ there is an element $d \in G$ such that $a * d = e$ and $d * a = e$.*

A group G is said to be abelian *if $a * b = b * a$ for all $a, b \in G$.*

The general group operation is usually denoted as $*$. Some groups have ordinary addition as its operation, in which case one writes the group operation as $+$. Various groups have ordinary multiplication as its operation. In that case, one usually writes ab instead of $a \times b$ where a and b are group elements.

The group axioms guarantee the existence of an identity element in addition to the existence of an inverse of an element. One can apply those axioms to show that any group element has unique identity and inverse.

Theorem 4.7. *Let G be a group and let $a, b, c \in G$. Then*

1. *G has a unique identity element.*
2. *Cancellation: If $a * b = a * c$ then $b = c$. Similarly, if $b * a = c * a$ then $b = c$.*
3. *Each element of G has a unique inverse.*

Proof. (1) Let e_1 and e_2 be identity elements in G. We show that these two identities are the same. We have $e_1 * e_2 = e_1$ and $e_2 * e_1 = e_2 = e_1 * e_2$, from which we get $e_1 = e_1 * e_2 = e_2$ so that there is only one identity element.

(2) Use the group axioms to see that each $a \in G$ has at least one inverse $d \in G$ such that $a * d = e = d * a$, where $e \in G$ is the unique identity element. If $a * b = a * c$ then $d * (a * b) = d * (a * c)$. By associativity and the properties of inverses and identity, we have the following implications

$$\begin{aligned} & (d * a) * b &=& (d * a) * c \\ \Longrightarrow \quad & e * b &=& c * e \\ \Longrightarrow \quad & b &=& c. \end{aligned}$$

The case where $b * a = c * a$ is proved by a similar argument.

(3) Suppose d_1 and d_2 are inverses of some $a \in G$. We show that these two inverses are equivalent. We have $a * d_1 = e = a * d_2$ and using the left cancellation property we get $d_1 = d_2$. Therefore a has a unique inverse. □

Definition 4.8. ***Multiplicative inverse.*** *Let $a, b \in \mathbf{Z}/n\mathbf{Z}$ where multiplication modulo n is the operation on elements of $\mathbf{Z}/n\mathbf{Z}$. Then b is a* multiplicative inverse *of a if $ab \equiv 1 \pmod{n}$. We also denote a multiplicative inverse of a by a^{-1}.*

Corollary 4.9. *An element $a \in \mathbf{Z}/n\mathbf{Z}$ has a multiplicative inverse in $\mathbf{Z}/n\mathbf{Z}$ if and only if $\gcd(a, n) = 1$.*

Proof. This follows from Theorem 4.5 by setting $b = 1$. □

Specializing the set $\mathbf{Z}/p\mathbf{Z}$ to the case where p is prime, each non-zero $e \in \mathbf{Z}/p\mathbf{Z}$ has a multiplicative inverse in $\mathbf{Z}/p\mathbf{Z}$. To see why this holds true, note that since $\gcd(a, p) = 1$ for any non-zero $a \in \mathbf{Z}/p\mathbf{Z}$, we can apply Bézout's identity to obtain $1 = am + pn$ for some $m, n \in \mathbf{Z}$. Solve the last equation for np to get $am - 1 = np$. Conclude by the definition of congruence to obtain $am \equiv 1 \pmod{p}$, and by Definition 4.8 we see that m is a multiplicative inverse of a modulo p.

Theorem 4.10. *Let $a, b, c, d, n \in \mathbf{Z}$ such that $n > 0$. If $a \equiv b \pmod{n}$ and $c \equiv d \pmod{n}$ then $ac \equiv bd \pmod{n}$.*

Proof. By definition of congruence we have $a - b = np$ and $c - d = nq$ for some $p, q \in \mathbf{Z}$. Consider the equation

$$n(pc + qb) = npc + nqb = (a - b)c + b(c - d)$$

hence $n \mid \big((a-b)c+b(c-d)\big)$. However, $(a-b)c+b(c-d) = ac-bc+bc-bd = ac-bd$, implying that $n \mid (ac-bd)$ or equivalently $ac-bd = n(pc+qb)$. Conclude by definition of congruence that $ac \equiv bd \pmod{n}$ as required. □

We now apply Corollary 4.9 to analyze the multiplicative group of $\mathbf{Z}/n\mathbf{Z}$.

Theorem 4.11. ***Multiplicative group.*** *Let $(\mathbf{Z}/n\mathbf{Z})^* = \{a \in \mathbf{Z}/n\mathbf{Z} \mid \gcd(a, n) = 1\}$. Then $(\mathbf{Z}/n\mathbf{Z})^*$ is an abelian group with respect to multiplication modulo n.*

Proof. First, note that multiplication modulo n is commutative since multiplication on the integers is commutative.

If $a, b \in (\mathbf{Z}/n\mathbf{Z})^*$, then by Corollary 4.9 both a and b have multiplicative inverses, say $aa^{-1} \equiv 1 \pmod{n}$ and $bb^{-1} \equiv 1 \pmod{n}$. Apply Theorem 4.10 to get $(aa^{-1})(bb^{-1}) \equiv 1 \pmod{n}$ so that $(a^{-1}b^{-1})(ab) \equiv 1 \pmod{n}$ by commutativity of multiplication modulo n as discussed in the first paragraph. Thus $a^{-1}b^{-1}$ is a multiplicative inverse of ab and therefore $ab \in (\mathbf{Z}/n\mathbf{Z})^*$ by Corollary 4.9.

Associativity of multiplication modulo n follows from associativity of multiplication on the integers. An identity element is 1. Finally, by definition of $(\mathbf{Z}/n\mathbf{Z})^*$ and Corollary 4.9, we see that each element of $(\mathbf{Z}/n\mathbf{Z})^*$ has a multiplicative inverse. Conclude by Definition 4.6 that $(\mathbf{Z}/n\mathbf{Z})^*$ is an abelian group with respect to multiplication modulo n. □

The abelian group $(\mathbf{Z}/n\mathbf{Z})^*$ in Theorem 4.11 is often referred to as the multiplicative group of $\mathbf{Z}/n\mathbf{Z}$. Its structure, and especially the number of elements it contains, can be analyzed via a number theoretic function called the Euler phi function.

Definition 4.12. ***Euler phi function.*** *If $n \in \mathbf{Z}$ is positive, then the* Euler phi function *of n, denoted $\varphi(n)$, counts the number of integers $0 < a < n$ such that $\gcd(a, n) = 1$. In symbols, we have*

$$\varphi(n) = \sum_{\substack{0<a<n \\ \gcd(a,n)=1}} 1.$$

The Euler phi function is also known as the Euler totient function.

If p is prime, then the number of elements in $(\mathbf{Z}/p\mathbf{Z})^*$ is given by the formula in Corollary 4.13. This result follows from the definitions of primes and the Euler phi function.

Corollary 4.13. *For any prime $p > 1$, we have $\varphi(p) = p - 1$.*

4.3 Encryption and decryption functions

In the affine cryptosystem, a secret key k is an ordered pair $k = (a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$. Similarly to the shift cryptosystem, the affine cryptosystem has $\mathbf{Z}/n\mathbf{Z}$ as both the plaintext space and the ciphertext space. The encryption function

$$\mathcal{E} : \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z} \longrightarrow \mathbf{Z}/n\mathbf{Z}$$

of the affine cryptosystem takes a secret key $k = (a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ and a plaintext element $p \in \mathbf{Z}/n\mathbf{Z}$ to produce a corresponding ciphertext element $c \in \mathbf{Z}/n\mathbf{Z}$ as given by

$$\mathcal{E}(k, p) = ap + b \pmod{n}.$$

That is

$$c \equiv ap + b \pmod{n}. \tag{4.3}$$

In order to solve the congruence equation (4.3) for p, we require that a has a multiplicative inverse modulo n. If $(a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ is a secret key, then each $b \in \mathbf{Z}/n\mathbf{Z}$ has a unique inverse with respect to addition modulo n, i.e. $n - b$ or $-b \pmod{n}$. Restricting a to be elements of the multiplicative group $(\mathbf{Z}/n\mathbf{Z})^*$ guarantees that a has a unique multiplicative inverse modulo n. Thus, the key space is $(\mathbf{Z}/n\mathbf{Z})^* \times \mathbf{Z}/n\mathbf{Z}$ and so the encryption function can be written as

$$\mathcal{E} : (\mathbf{Z}/n\mathbf{Z})^* \times \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z} \longrightarrow \mathbf{Z}/n\mathbf{Z}.$$

By definition of the Euler phi function, $(\mathbf{Z}/n\mathbf{Z})^*$ consists of $\varphi(n)$ elements. Therefore the key space consists of $\varphi(n) \times n$ possible keys.

To see that the encryption function $\mathcal{E}$ is bijective, we require any linear congruence of the form $ax + b \equiv y \pmod{n}$ to have a unique solution for x. This congruence is equivalent to $ax \equiv y - b \pmod{n}$. Hence it suffices to require that any linear congruence of the form $ax \equiv y \pmod{n}$ has a unique solution for x. By Theorem 4.5 the required unique solution exists if and only if $\gcd(a, n) = 1$. This is certainly the case with the multiplicative group $(\mathbf{Z}/n\mathbf{Z})^*$.

By the choice of the key space, note that the decryption function

$$\mathcal{D} : (\mathbf{Z}/n\mathbf{Z})^* \times \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z} \longrightarrow \mathbf{Z}/n\mathbf{Z}$$

is the inverse function of $\mathcal{E}$. Solve the congruence in (4.3) for p to obtain the corresponding decryption function:

$$\mathcal{D}(k, c) = a^{-1}(c - b) \pmod{n}.$$

That is

$$p \equiv a^{-1}(c - b) \pmod{n}. \tag{4.4}$$

The inverse key corresponding to a key $(a, b) \in (\mathbf{Z}/n\mathbf{Z})^* \times \mathbf{Z}/n\mathbf{Z}$ can be obtained as follows. The right-hand side of (4.4) contains the expression $a^{-1}(c-b) \pmod{n}$, which evaluates to an element in $\mathbf{Z}/n\mathbf{Z}$. Distribute a^{-1} to get $a^{-1}c - a^{-1}b \pmod{n}$. Thus if $k = (a, b)$ is a secret key, then the inverse key k^{-1} corresponding to k is

$$k^{-1} = (a^{-1}, -a^{-1}b). \tag{4.5}$$

We have implemented the affine cryptosystem in the class

`sage.crypto.classical.AffineCryptosystem`

of the Sage standard library. The full reference manual of our implementation is contained in Appendix B.

In general, the expression (4.5) for inverse keys allows for expressing both $\mathcal{E}$ and $\mathcal{D}$ as a function of the general form

$$\mathcal{F}(K, m) = Ac + B \pmod{n}.$$

If $K = (a, b)$ is a secret key and m is a plaintext element, then $A = a$ and $B = b$ so that $\mathcal{F}(K, m)$ defines the encryption function $\mathcal{E}$. Where $K = (a^{-1}, -a^{-1}b)$ is the inverse key corresponding to a secret key and m is a ciphertext element, then $A = a^{-1}$ and $B = -a^{-1}b$ so that $\mathcal{F}(K, m)$ defines the decryption function $\mathcal{D}$.

4.4 Cryptanalysis

The affine cryptosystem is vulnerable to all of the attacks described in section 3.4. Recall that a shift cryptosystem with key space $\mathbf{Z}/n\mathbf{Z}$ has at most n possible unique keys. In generalizing the shift cryptosystem, the affine cryptosystem increases the possible number of unique keys by a factor of $\varphi(n)$, resulting in a total of $\varphi(n) \times n$ possible unique keys. However, the size of the key space of the affine cryptosystem is still small enough that an exhaustive key search attack is feasible. We can also apply the technique of monogram frequency analysis (see section 3.4.2) to cryptanalyze the affine cryptosystem. Another cryptanalytic attack is to combine exhaustive key search with monogram frequency analysis (as described in section 3.4.3) in order to rank all of the $\varphi(n) \times n$ keys.

The brute-force attack against the affine cryptosystem is implemented in the method

`sage.crypto.classical.AffineCryptosystem.brute_force`

of the Sage standard library. The two key ranking functions are implemented in the methods

`sage.crypto.classical.AffineCryptosystem.rank_by_chi_square`
`sage.crypto.classical.AffineCryptosystem.rank_by_squared_differences`

which are also part of the Sage standard library. See sections B.2.1, B.2.7 and B.2.8, respectively, for the reference manual of our implementation.

4.5 Example Sage usage

This section provides some examples on using the functionalities in our implementation of the affine cryptosystem. Refer to Appendix B for the full reference manual. Our Sage implementation supports encryption and decryption over the capital letters of the English alphabet:

```
sage: A = AffineCryptosystem(AlphabeticStrings()); A
Affine cryptosystem on Free alphabetic string monoid on A-Z
sage: P = A.encoding("The affine cryptosystem generalizes the shift cipher.")
sage: P
THEAFFINECRYPTOSYSTEMGENERALIZESTHESHIFTCIPHER
sage: a, b = (9, 13)
sage: C = A.enciphering(a, b, P); C
CYXNGGHAXFKVSCJTVTCXRPXAXKNIHEXTCYXTYHGCFHSYXK
sage: A.deciphering(a, b, C)
THEAFFINECRYPTOSYSTEMGENERALIZESTHESHIFTCIPHER
sage: A.deciphering(a, b, C) == P
True
```

We can also use functional notation to work through the previous example:

```
sage: A = AffineCryptosystem(AlphabeticStrings()); A
Affine cryptosystem on Free alphabetic string monoid on A-Z
sage: P = A.encoding("The affine cryptosystem generalizes the shift cipher.")
sage: P
THEAFFINECRYPTOSYSTEMGENERALIZESTHESHIFTCIPHER
sage: a, b = (9, 13)
sage: E = A(a, b); E
Affine cipher on Free alphabetic string monoid on A-Z
sage: C = E(P); C
CYXNGGHAXFKVSCJTVTCXRPXAXKNIHEXTCYXTYHGCFHSYXK
sage: aInv, bInv = A.inverse_key(a, b)
sage: D = A(aInv, bInv); D
Affine cipher on Free alphabetic string monoid on A-Z
sage: D(C)
THEAFFINECRYPTOSYSTEMGENERALIZESTHESHIFTCIPHER
sage: D(C) == P
True
sage: D(C) == P == D(E(P))
True
```

Encrypting the ciphertext with the inverse key also produces the plaintext:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: P = A.encoding("Encrypt with inverse key.")
sage: a, b = (11, 8)
sage: C = A.enciphering(a, b, P)
sage: P; C
ENCRYPTWITHINVERSEKEY
AVENMRJQSJHSVFANYAOAM
sage: aInv, bInv = A.inverse_key(a, b)
```

```
sage: A.enciphering(aInv, bInv, C)
ENCRYPTWITHINVERSEKEY
sage: A.enciphering(aInv, bInv, C) == P
True
```

For a secret key $(a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$, if $a = 1$ then any affine cryptosystem with key $(1, b)$ for any $b \in \mathbf{Z}/n\mathbf{Z}$ is a shift cryptosystem. Here is how one can create a Caesar cipher using the affine cryptosystem:

```
sage: caesar = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (1, 3)
sage: P = caesar.encoding("abcdef"); P
ABCDEF
sage: C = caesar.enciphering(a, b, P); C
DEFGHI
sage: caesar.deciphering(a, b, C) == P
True
```

An affine cryptosystem with keys of the form $(a, 0) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ is called a decimation cipher on the Roman alphabet, or decimation cipher for short:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: P = A.encoding("A decimation cipher is a specialized affine cipher.")
sage: a, b = (17, 0)
sage: C = A.enciphering(a, b, P)
sage: P; C
ADECIMATIONCIPHERISASPECIALIZEDAFFINECIPHER
AZQIGWALGENIGVPQDGUAUVQIGAFGJQZAHHGNQIGVPQD
sage: A.deciphering(a, b, C) == P
True
```

Generate a random key for encryption and decryption:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: P = A.encoding("An affine cipher with a random key.")
sage: a, b = A.random_key()
sage: C = A.enciphering(a, b, P)
sage: A.deciphering(a, b, C) == P
True
```

We can cryptanalyze using the technique of exhaustive key search:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (3, 7)
sage: P = A.encoding("Linear"); P
LINEAR
sage: C = A.enciphering(a, b, P)
sage: L = A.brute_force(C); len(L)  # the number of candidate decipherments
312
sage: sorted(L.items())[:26]  # display the first 26 candidate decipherments
<BLANKLINE>
[((1, 0), OFUTHG),
((1, 1), NETSGF),
((1, 2), MDSRFE),
((1, 3), LCRQED),
((1, 4), KBQPDC),
((1, 5), JAPOCB),
((1, 6), IZONBA),
```

```
((1, 7), HYNMAZ),
((1, 8), GXMLZY),
((1, 9), FWLKYX),
((1, 10), EVKJXW),
((1, 11), DUJIWV),
((1, 12), CTIHVU),
((1, 13), BSHGUT),
((1, 14), ARGFTS),
((1, 15), ZQFESR),
((1, 16), YPEDRQ),
((1, 17), XODCQP),
((1, 18), WNCBPO),
((1, 19), VMBAON),
((1, 20), ULAZNM),
((1, 21), TKZYML),
((1, 22), SJYXLK),
((1, 23), RIXWKJ),
((1, 24), QHWVJI),
((1, 25), PGVUIH)]
```

For a short sample of ciphertext, the chi-square ranking function is more effective than the squared-differences function:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (19, 4)
sage: P = A.encoding("Line"); P
LINE
sage: C = A.enciphering(a, b, P)
sage: Rank = A.brute_force(C, ranking="chisquare")
sage: Rank[:5]  # display only the top 5 candidate keys
<BLANKLINE>
[((15, 18), NETS),
 ((19, 4), LINE),
 ((21, 17), STAD),
 ((23, 7), SLOT),
 ((23, 0), HADI)]
sage: Rank = A.brute_force(C, ranking="squared_differences")
sage: Rank[:5]  # display only the top 5 candidate keys
<BLANKLINE>
[((15, 18), NETS),
 ((17, 15), ETUN),
 ((1, 24), HCTE),
 ((19, 4), LINE),
 ((21, 20), DELO)]
```

As the ciphertext sample increases, both the chi-square and squared-differences ranking functions have similar performance:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (3, 7)
sage: P = A.encoding("Linear functions for encrypting and decrypting."); P
LINEARFUNCTIONSFORENCRYPTINGANDDECRYPTING
sage: C = A.enciphering(a, b, P)
sage: Rank = A.brute_force(C, ranking="chisquare")
sage: Rank[:10]  # display only the top 10 candidate keys
<BLANKLINE>
[((3, 7), LINEARFUNCTIONSFORENCRYPTINGANDDECRYPTING),
((23, 25), VYTCGPBMTENYSTOBSPCTEPIRNYTAGTDDCEPIRNYTA),
((1, 12), CTIHVUKDIBATLIXKLUHIBUPOATINVIEEHBUPOATIN),
((11, 15), HSRYELDAROVSWRQDWLYROLUBVSRIERTTYOLUBVSRI),
((25, 1), NWHIUVFMHOPWEHSFEVIHOVABPWHCUHLLIOVABPWHC),
((25, 7), TCNOABLSNUVCKNYLKBONUBGHVCNIANRROUBGHVCNI),
((15, 4), SHIBVOWZILEHDIJWDOBILOFYEHIRVIGGBLOFYEHIR),
((15, 23), PEFYSLTWFIBEAFGTALYFILCVBEFOSFDDYILCVBEFO),
```

```
((7, 10), IDUFHSYXUTEDNULYNSFUTSVGEDURHUMMFTSVGEDUR),
((19, 22), QVETRGABEFUVLENALGTEFGDSUVEHREMMTFGDSUVEH)]
sage: Rank = A.brute_force(C, ranking="squared_differences")
sage: Rank[:10]  # display only the top 10 candidate keys
<BLANKLINE>
[((3, 7), LINEARFUNCTIONSFORENCRYPTINGANDDECRYPTING),
((23, 6), GJENRAMXEPYJDEZMDANEPATCYJELREOONPATCYJEL),
((23, 25), VYTCGPBMTENYSTOBSPCTEPIRNYTAGTDDCEPIRNYTA),
((19, 22), QVETRGABEFUVLENALGTEFGDSUVEHREMMTFGDSUVEH),
((19, 9), DIRGETNORSHIYRANYTGRSTQFHIRUERZZGSTQFHIRU),
((23, 18), KNIRVEQBITCNHIDQHERITEXGCNIPVISSRTEXGCNIP),
((17, 16), GHORBEIDOJMHFOVIFEROJETWMHOZBOAARJETWMHOZ),
((21, 14), AHEZRMOFEVQHTEBOTMZEVMNIQHEDREKKZVMNIQHED),
((1, 12), CTIHVUKDIBATLIXKLUHIBUPOATINVIEEHBUPOATIN),
((7, 18), SNEPRCIHEDONXEVIXCPEDCFQONEBREWWPDCFQONEB)]
```

Chapter 5

Simplified Data Encryption Standard

The Data Encryption Standard (DES) is a symmetric-key cryptosystem developed at IBM. In 1977, the National Bureau of Standards (which is currently the National Institute of Standards and Technology, or NIST) adopted DES as an encryption standard for all "unclassified" applications. The official description of DES originally appeared in the Federal Information Processing Standards (FIPS) Publication 46 [1], dated 15th January 1977. The standard has since been reaffirmed a total of five times, each in 1983, 1988, 1993 and 1999. The last reaffirmation of DES is dated 25th October 1999 as FIPS 46-3 [2]. On 19th May 2005, NIST published an announcement [86] in the Federal Register to withdraw FIPS 46-3 as well as FIPS 74 Guidelines for Implementing and Using the NBS Data Encryption Standard, and FIPS 81 DES Modes of Operation.

The full DES algorithm operates on 64-bit blocks of ciphertext/plaintext with a 56-bit secret key, making the algorithm itself unsuitable for working through by hand in order to understand its general structure. This chapter describes a simplified version of DES designed by Schaefer [102] and named as simplified DES or S-DES. S-DES is a version of DES with all parameters significantly reduced, but at the same time preserving the structure of DES. A primary goal of S-DES is to allow students of cryptology to understand the general structure of DES, thus laying a foundation for a thorough study of DES itself.

We begin in section 5.1 with a description of the key space of S-DES together with the process whereby subkeys can be generated from an S-DES secret key. Section 5.2 describes the encryption and decryption functions of S-DES. Along the way, the permutation and round functions that form the heart of S-DES are also specified. The specification of S-DES as presented in this chapter has been implemented as part of the Sage [111] standard library. Full source code of our implementation is available with the latest stable release of Sage, which as of this writing is Sage version 4.2.1. Refer to Appendix C for the reference manual of our implementation. Finally, in section 5.3 we provide numerous examples that illustrate functionalities of our implementation.

5.1 The S-DES secret keys

Denote by

$$(\mathbf{Z}/n\mathbf{Z})^k = \underbrace{\mathbf{Z}/n\mathbf{Z} \times \cdots \times \mathbf{Z}/n\mathbf{Z}}_{k \text{ copies}}$$

the Cartesian product of k copies of $\mathbf{Z}/n\mathbf{Z}$. The S-DES encryption and decryption algorithms operate on 8-bit blocks of ciphertext/plaintext. Let $k = (k_0, k_1, k_2, \ldots, k_9)$ be a secret key where each $k_i \in \mathbf{Z}/2\mathbf{Z}$. It follows that there are $2^{10} = 1024$ possible keys within the key space $(\mathbf{Z}/2\mathbf{Z})^{10}$. The secret key is a 10-bit block from which two subkeys are derived.

The process of deriving the two subkeys corresponding to a secret key k involves various permutation and shifting functions, which we now describe. Let $P_{10} : (\mathbf{Z}/2\mathbf{Z})^{10} \longrightarrow (\mathbf{Z}/2\mathbf{Z})^{10}$ be a permutation on a 10-bit block defined by

$$P_{10}(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9) = (b_2, b_4, b_1, b_6, b_3, b_9, b_0, b_8, b_7, b_5). \tag{5.1}$$

Similarly, define a function $P_8 : (\mathbf{Z}/2\mathbf{Z})^{10} \longrightarrow (\mathbf{Z}/2\mathbf{Z})^8$ that takes a 10-bit block, extracts 8 bits from the input block, and permutes the resulting 8 bits:

$$P_8(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9) = (b_5, b_2, b_6, b_3, b_7, b_4, b_9, b_8). \tag{5.2}$$

The function in (5.2) is not strictly a permutation since it is not bijective. However, we shall refer to P_8 as a permutation. Note that the permutation P_8 excludes the bits b_0 and b_1 from the final permuted sub-block. The permutations P_{10} and P_8 are illustrated in Figures 5.1 and 5.2 respectively.

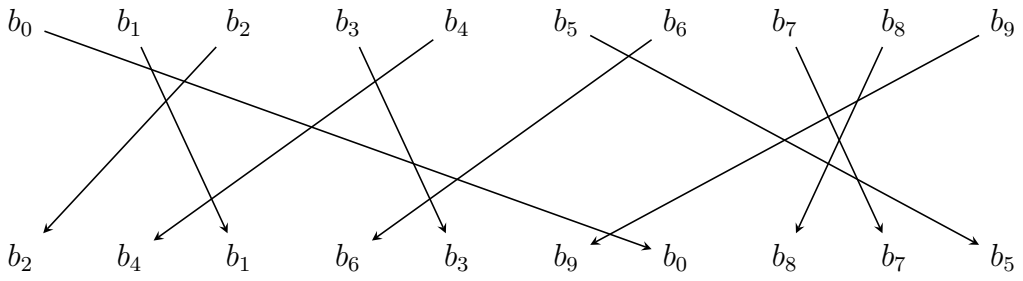

Figure 5.1: The S-DES permutation P_{10}.

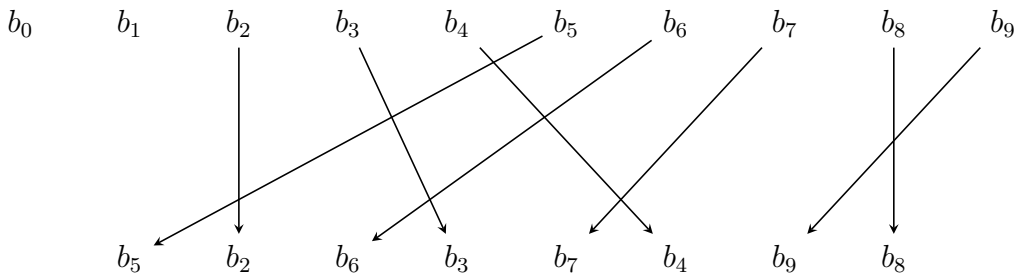

Figure 5.2: The S-DES permutation P_8.

Define a shifting function $L_n : (\mathbf{Z}/2\mathbf{Z})^8 \longrightarrow (\mathbf{Z}/2\mathbf{Z})^8$ that partitions an 8-bit input block into two halves: a left half $B_1 = (b_0, b_1, b_2, b_3, b_4)$ and a right half $B_2 = (b_5, b_6, b_7, b_8, b_9)$. The function L_n then performs a left shift of n positions on each B_i with wrap around. That is, if $n = 1$ then

$$L_1(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9) = (b_1, b_2, b_3, b_4, b_0, b_6, b_7, b_8, b_9, b_5) \tag{5.3}$$

and for $n = 2$ we have

$$L_2(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9) = (b_2, b_3, b_4, b_0, b_1, b_7, b_8, b_9, b_5, b_6). \tag{5.4}$$

The S-DES algorithm specifies only two left-shift functions, i.e. L_1 and L_2. These two left shift functions are illustrated in Figures 5.3 and 5.4 respectively.

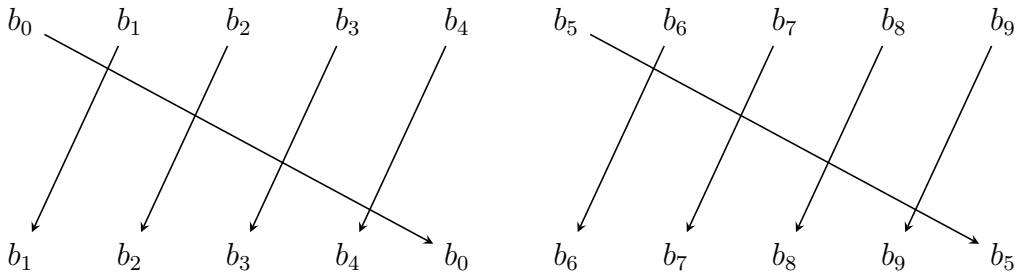

Figure 5.3: The left-shift function L_1.

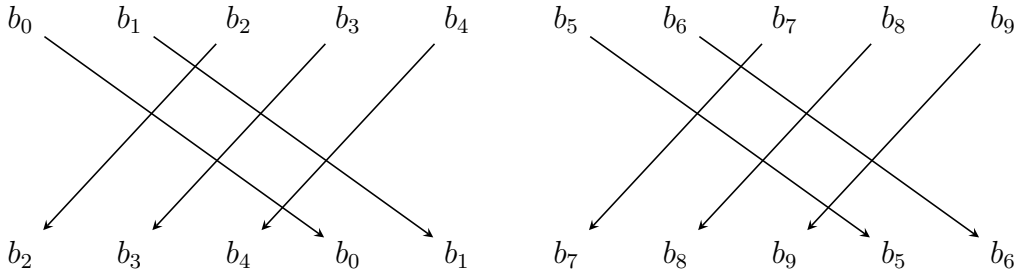

Figure 5.4: The left-shift function L_2.

Now we are ready to define the subkeys. The first subkey K_1 is given by

$$K_1(k) = P_8(L_1((P_{10}(k)))) \tag{5.5}$$

which can be described as the function composition $K_1 = P_8 \circ L_1 \circ P_{10}$ where the order of execution is from right to left. The second subkey K_2 is defined as

$$K_2(k) = P_8(L_2((L_1(P_{10}(k))))) \tag{5.6}$$

which is the function composition $K_1 = P_8 \circ L_2 \circ L_1 \circ P_{10}$. Again, the order of execution is from right to left. The processes of generating subkeys K_1 and K_2 are illustrated in Figures 5.5 and 5.6 respectively.

We have implemented S-DES in the class

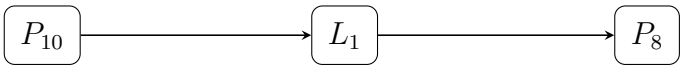

Figure 5.5: Generating subkey K_1.

`sage.crypto.block_cipher.sdes.SimplifiedDES`

of the Sage standard library. The reference manual of our implementation can be found in Appendix C. The procedure for deriving the two subkeys from a secret key is implemented in the method

`sage.crypto.block_cipher.sdes.SimplifiedDES.subkey`

whose reference manual is contained in section C.2.14.

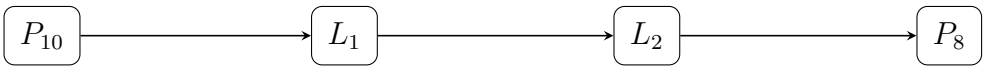

Figure 5.6: Generating subkey K_2.

5.2 Encryption and decryption functions

The encryption and decryption functions of S-DES each takes a 10-bit secret key and an 8-bit block as input, and produces an 8-bit block as output. Thus there is a total of $2^8 = 256$ different input and output blocks. Given a plaintext $\mathbf{P} = (p_0, p_1, p_2, \ldots, p_{m-1})$ of m bits such that $8 \mid m$, $\mathbf{P}$ is first split into $m/8$ blocks each of which contains 8 bits. Identical blocks of input result in identical blocks of output.

The S-DES encryption function

$$\mathcal{E} : (\mathbf{Z}/2\mathbf{Z})^{10} \times (\mathbf{Z}/2\mathbf{Z})^8 \longrightarrow (\mathbf{Z}/2\mathbf{Z})^8$$

and its corresponding decryption function

$$\mathcal{D} : (\mathbf{Z}/2\mathbf{Z})^{10} \times (\mathbf{Z}/2\mathbf{Z})^8 \longrightarrow (\mathbf{Z}/2\mathbf{Z})^8$$

can each be described as a composition of five other functions: an initial permutation P and its inverse permutation P^{-1}, two Feistel round functions Π_{F,K_1} and Π_{F,K_2} operating on the first and second subkeys respectively, and a bit switching function σ. Using these five maps, $\mathcal{E}$ and $\mathcal{D}$ are given by the following function compositions

$$\mathcal{E} = P^{-1} \circ \Pi_{F,K_2} \circ \sigma \circ \Pi_{F,K_1} \circ P \tag{5.7}$$
$$\mathcal{D} = P^{-1} \circ \Pi_{F,K_1} \circ \sigma \circ \Pi_{F,K_2} \circ P \tag{5.8}$$

where the order of execution is from right to left. We now describe each of the five individual functions in (5.7) and (5.8) whose composition in various order define $\mathcal{E}$ and $\mathcal{D}$.

The above encryption and decryption functions are implemented in the methods

`sage.crypto.block_cipher.sdes.SimplifiedDES.encrypt`
`sage.crypto.block_cipher.sdes.SimplifiedDES.decrypt`

of the Sage standard library. Refer to sections C.2.2 and C.2.3, respectively, for the reference manual of these two methods.

5.2.1 Permutation and switch functions

Similarly to the derivation of the subkeys, $\mathcal{E}$ relies on a number of functions that permute the bits of an input block. One such permutation function is the initial permutation function

$$P : (\mathbf{Z}/2\mathbf{Z})^8 \longrightarrow (\mathbf{Z}/2\mathbf{Z})^8$$

which permutes the eight bits of an input block as follows:

$$P(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7) = (b_1, b_5, b_2, b_0, b_3, b_7, b_4, b_6). \tag{5.9}$$

From (5.9), we see that the inverse permutation $P^{-1} : (\mathbf{Z}/2\mathbf{Z})^8 \longrightarrow (\mathbf{Z}/2\mathbf{Z})^8$ is

$$P^{-1}(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7) = (b_3, b_0, b_2, b_4, b_6, b_1, b_7, b_5). \tag{5.10}$$

The initial permutation function (5.9) and its inverse permutation (5.10) are illustrated in Figure 5.7. Their implementation is contained in the method

`sage.crypto.block_cipher.sdes.SimplifiedDES.initial_permutation`

whose reference manual is given in section C.2.4.

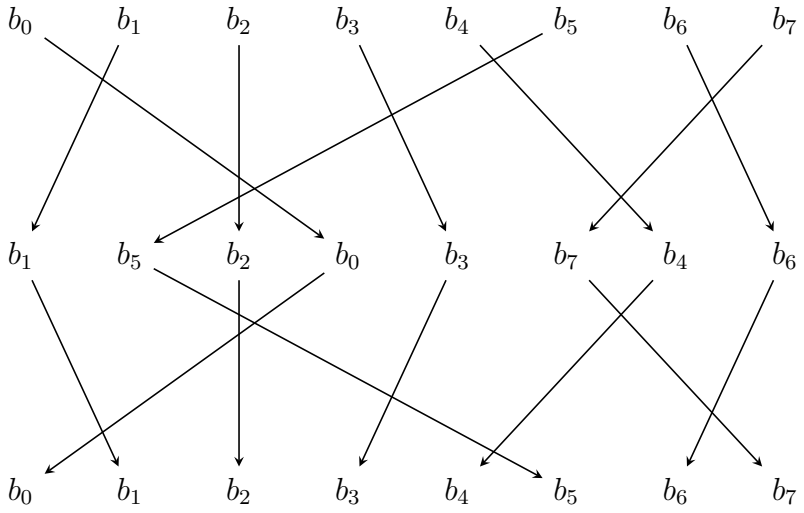

Figure 5.7: The initial permutation and its inverse.

However, unlike the subkey derivation process, $\mathcal{E}$ uses a switch function

$$\sigma : (\mathbf{Z}/2\mathbf{Z})^8 \longrightarrow (\mathbf{Z}/2\mathbf{Z})^8$$

instead of a left-shift function. The switch function σ interchanges the first 4 bits of an input block with the last 4 bits of the input block. That is,

$$\sigma(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7) = (b_4, b_5, b_6, b_7, b_0, b_1, b_2, b_3). \tag{5.11}$$

In other words, σ first partitions its input block into two sub-blocks of equal length, and then proceeds to switch those sub-blocks around. The sub-block switching function (5.11) is illustrated in Figure 5.8. Our Sage implementation is contained in the method

`sage.crypto.block_cipher.sdes.SimplifiedDES.switch`

and the reference manual is given in section C.2.15.

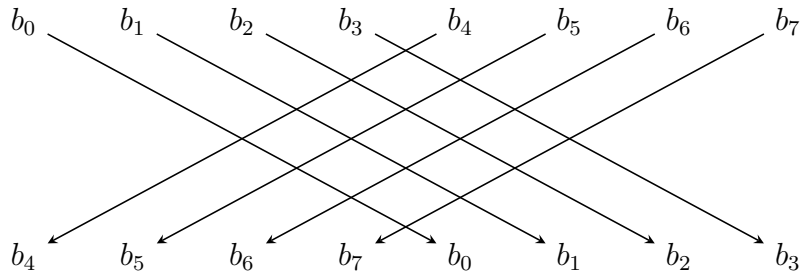

Figure 5.8: The sub-block switch function.

5.2.2 The Feistel round function

The function

$$\Pi_{F,K_i} : (\mathbf{Z}/2\mathbf{Z})^8 \times (\mathbf{Z}/2\mathbf{Z})^8 \longrightarrow (\mathbf{Z}/2\mathbf{Z})^8 \tag{5.12}$$

is a Feistel-like round function that mimics the structure of the round function in the full DES algorithm. See FIPS 46-3 [2] for a complete description of the DES round function or refer to cryptography texts such as Mollin [68], Stinson [113], or Trappe and Washington [114]. The Feistel function (5.12) relies on two substitution boxes, or S-boxes, called S_0 and S_1 as shown in Tables 5.1 and 5.2 respectively.

Input	Output	Input	Output
0000	01	1000	00
0001	00	1001	10
0010	11	1010	01
0011	10	1011	11
0100	11	1100	11
0101	10	1101	01
0110	01	1110	11
0111	00	1111	10

Table 5.1: The S-box S_0 of simplified DES.

Let $b = (b_0, b_1, b_2, \ldots, b_7)$ be a block of 8 bits where each $b_i \in \mathbf{Z}/2\mathbf{Z}$, let L and R be the left-most 4 bits and right-most 4 bits of b respectively, and let $F : (\mathbf{Z}/2\mathbf{Z})^4 \times (\mathbf{Z}/2\mathbf{Z})^8 \longrightarrow (\mathbf{Z}/2\mathbf{Z})^4$ be a function mapping a 4-bit block and an 8-bit subkey to a 4-bit output. Then

$$\Pi_{F,K_i}(L, R) = \big(L \oplus F(R, K_i),\, R\big)$$

Input	Output	Input	Output
0000	00	1000	11
0001	01	1001	00
0010	10	1010	01
0011	11	1011	00
0100	10	1100	10
0101	00	1101	01
0110	01	1110	00
0111	11	1111	11

Table 5.2: The S-box S_1 of simplified DES.

where K_i is the i-th subkey and $\oplus$ denotes addition in $\mathbf{Z}/2\mathbf{Z}$. One can also interpret $\oplus$ as the operation of bit-wise exclusive-or. Figure 5.9 illustrates the round function Π_{F,K_i}.

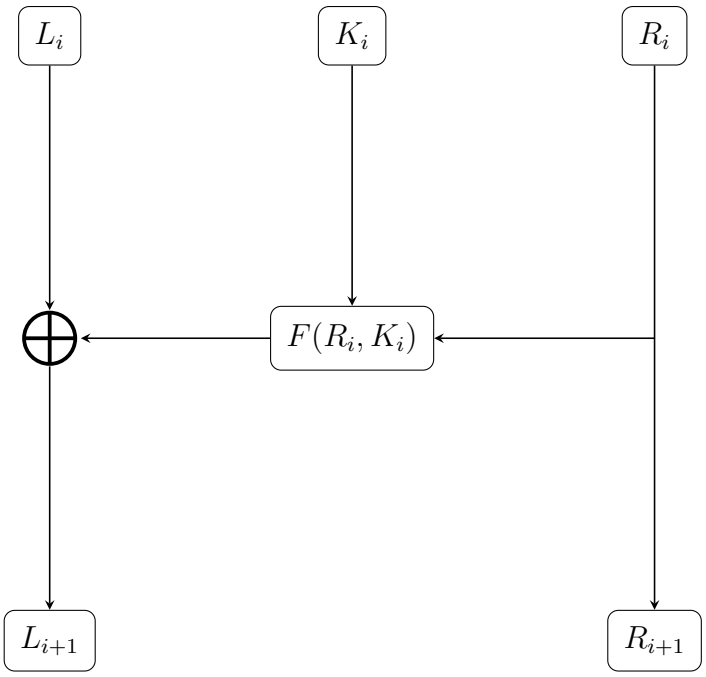

Figure 5.9: The Feistel round function Π_{F,K_i}.

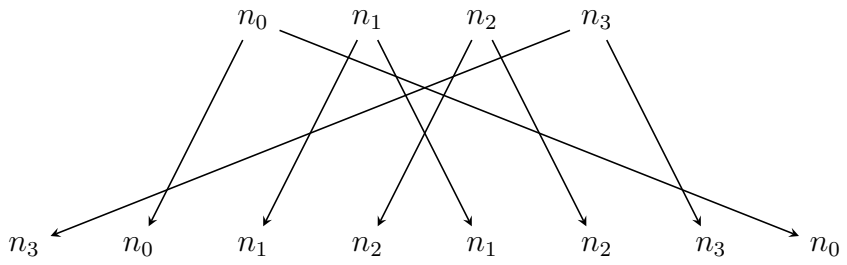

Figure 5.10: The expansion function E.

The mixing function F can be described as follows. Consider an expansion function $E : (\mathbf{Z}/2\mathbf{Z})^4 \longrightarrow (\mathbf{Z}/2\mathbf{Z})^8$ that takes a 4-bit block $B = (n_0, n_1, n_2, n_3)$ and

expands it into an 8-bit block B^* according to the rule

$$E(n_0, n_1, n_2, n_3) = (n_3, n_0, n_1, n_2, n_1, n_2, n_3, n_0).$$

The expansion function is illustrated in Figure 5.10; the expanded block $B^* = (n_3, n_0, n_1, n_2, n_1, n_2, n_3, n_0)$ can be represented as

$$\begin{array}{c|cc|c} n_3 & n_0 & n_1 & n_2 \\ n_1 & n_2 & n_3 & n_0 \end{array}. \tag{5.13}$$

The 8-bit subkey $K_i = (k_0, k_1, k_2, k_3, k_4, k_5, k_6, k_7)$ of F is then added to (5.13) using addition in $\mathbf{Z}/2\mathbf{Z}$ to produce

$$\begin{array}{c|cc|c} n_3 + k_0 & n_0 + k_1 & n_1 + k_2 & n_2 + k_3 \\ n_1 + k_4 & n_2 + k_5 & n_3 + k_6 & n_0 + k_7 \end{array} = \begin{array}{c|cc|c} p_{0,0} & p_{0,1} & p_{0,2} & p_{0,3} \\ p_{1,0} & p_{1,1} & p_{1,2} & p_{1,3} \end{array}. \tag{5.14}$$

Now read the first row on the right-hand side of (5.14) as the 4-bit string $B_1 = p_{0,0}p_{0,3}p_{0,1}p_{0,2}$ and input this 4-bit string through S-box S_0 to obtain a 2-bit output. Next, read the second row on the right-hand side of (5.14) as the 4-bit string $B_2 = p_{1,0}p_{1,3}p_{1,1}p_{1,2}$ and input this 4-bit string through S-box S_1 to produce another 2-bit output. We now have a total of 4 bits as produced by S_0 and S_1. Denote these 4 bits as $b = (b_0, b_1, b_2, b_3)$ and input b into the permutation function $P_4 : (\mathbf{Z}/2\mathbf{Z})^4 \longrightarrow (\mathbf{Z}/2\mathbf{Z})^4$ given by

$$P_4(b_0, b_1, b_2, b_3) = (b_1, b_3, b_2, b_0). \tag{5.15}$$

We take the result of P_4 as the output of F. The inner working of the mixing function F is illustrated in Figure 5.12. The permutation function (5.15) is illustrated in Figure 5.11 and implemented as the Sage method

`sage.crypto.block_cipher.sdes.SimplifiedDES.permutation4`

whose reference manual is given in section C.2.8.

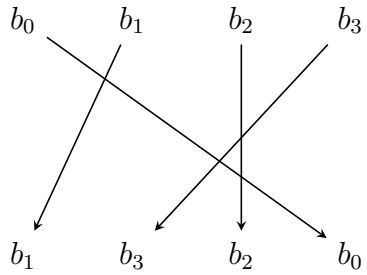

Figure 5.11: The permutation function P_4.

5.3 Example Sage usage

This section provides examples illustrating functionalities of our implementation of S-DES within the Sage [111] standard library. The reference manual of our implementation is contained in Appendix C and the source code is available with the latest stable release of Sage.

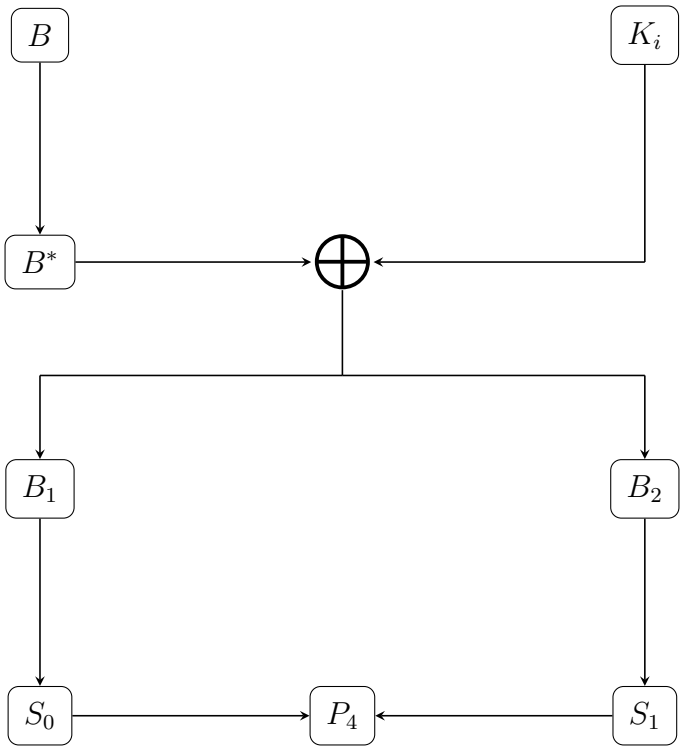

Figure 5.12: The mixing function F.

Our Sage implementation supports encryption of an 8-bit plaintext block. One can work with a list of 8 bits or a string of 8 bits:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: P = [0, 1, 0, 1, 0, 1, 0, 1]
sage: K = [1, 0, 1, 0, 0, 0, 0, 0, 1, 0]
sage: sdes.encrypt(P, K)
[1, 1, 0, 0, 0, 0, 0, 1]
sage: P = "01010101"
sage: K = "1010000010"
sage: sdes.encrypt(sdes.string_to_list(P), sdes.string_to_list(K))
[1, 1, 0, 0, 0, 0, 0, 1]
```

Similarly, decryption can be performed on an 8-bit plaintext block whose representation is as a list of 8 bits or as a string of 8 bits:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: C = [0, 1, 0, 1, 0, 1, 0, 1]
sage: K = [1, 0, 1, 0, 0, 0, 0, 0, 1, 0]
sage: sdes.decrypt(C, K)
[0, 0, 0, 1, 0, 1, 0, 1]
sage: C = "01010101"
sage: K = "1010000010"
sage: sdes.decrypt(sdes.string_to_list(C), sdes.string_to_list(K))
[0, 0, 0, 1, 0, 1, 0, 1]
```

We can encrypt a random block of 8-bit plaintext using a random key, decrypt the ciphertext, and compare the result with the original plaintext:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES(); sdes
Simplified DES block cipher with 10-bit keys
sage: bin = BinaryStrings()
sage: P = [bin(str(randint(0, 1))) for i in xrange(8)]
sage: K = sdes.random_key()
sage: C = sdes.encrypt(P, K)
sage: plaintxt = sdes.decrypt(C, K)
sage: plaintxt == P
True
```

We can also encrypt binary strings that are larger than 8 bits in length. However, the number of bits in that binary string must be positive and a multiple of 8:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: bin = BinaryStrings()
sage: P = bin.encoding("Encrypt this using S-DES!")
sage: Mod(len(P), 8) == 0
True
sage: K = sdes.list_to_string(sdes.random_key())
sage: C = sdes(P, K, algorithm="encrypt")
sage: plaintxt = sdes(C, K, algorithm="decrypt")
sage: plaintxt == P
True
```

Here is a demonstration of the various permutation functions of simplified DES:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: #
sage: # the initial permutation and its corresponding inverse permutation
sage: sdes = SimplifiedDES()
sage: B = [1, 0, 1, 1, 0, 1, 0, 0]
sage: P = sdes.initial_permutation(B); P
[0, 1, 1, 1, 1, 0, 0, 0]
sage: Pinv = sdes.initial_permutation(P, inverse=True)
sage: Pinv; B
[1, 0, 1, 1, 0, 1, 0, 0]
[1, 0, 1, 1, 0, 1, 0, 0]
sage: #
sage: # the permutation P_10
sage: B = [1, 1, 0, 0, 1, 0, 0, 1, 0, 1]
sage: sdes.permutation10(B)
[0, 1, 1, 0, 0, 1, 1, 0, 1, 0]
sage: sdes.permutation10([0, 1, 1, 0, 1, 0, 0, 1, 0, 1])
[1, 1, 1, 0, 0, 1, 0, 0, 1, 0]
sage: sdes.permutation10([1, 0, 1, 0, 0, 0, 0, 0, 1, 0])
[1, 0, 0, 0, 0, 0, 1, 1, 0, 0]
sage: #
sage: # the permutation P_4
sage: B = [1, 1, 0, 0]
sage: sdes.permutation4(B)
[1, 0, 0, 1]
sage: sdes.permutation4([0, 1, 0, 1])
[1, 1, 0, 0]
sage: #
sage: # the permutation P_8
sage: B = [1, 1, 0, 0, 1, 0, 0, 1, 0, 1]
sage: sdes.permutation8(B)
[0, 0, 0, 0, 1, 1, 1, 0]
sage: sdes.permutation8([0, 1, 1, 0, 1, 0, 0, 1, 0, 1])
[0, 1, 0, 0, 1, 1, 1, 0]
sage: sdes.permutation8([0, 0, 0, 0, 1, 1, 1, 0, 0, 0])
[1, 0, 1, 0, 0, 1, 0, 0]
```

We illustrate the operations of the left-shift function as well as the switch function:

```
sage: # Circular left shift by 1 position of a 10-bit string
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: B = [1, 0, 0, 0, 0, 0, 1, 1, 0, 0]
sage: sdes.left_shift(B)
[0, 0, 0, 0, 1, 1, 1, 0, 0, 0]
sage: sdes.left_shift([1, 0, 1, 0, 0, 0, 0, 0, 1, 0])
[0, 1, 0, 0, 1, 0, 0, 1, 0, 0]
sage: #
sage: # Circular left shift by 2 positions of a 10-bit string
sage: B = [0, 0, 0, 0, 1, 1, 1, 0, 0, 0]
sage: sdes.left_shift(B, n=2)
[0, 0, 1, 0, 0, 0, 0, 0, 1, 1]
sage: #
sage: # the switch function
sage: B = [1, 1, 1, 0, 1, 0, 0, 0]
sage: sdes.switch(B)
[1, 0, 0, 0, 1, 1, 1, 0]
sage: sdes.switch([1, 1, 1, 1, 0, 0, 0, 0])
[0, 0, 0, 0, 1, 1, 1, 1]
```

Generating a random key and producing the two subkeys of a secret key:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: #
sage: # generate a random key
sage: sdes = SimplifiedDES()
sage: key = sdes.random_key()
sage: len(key) == sdes.block_length()
True
sage: #
sage: # get the subkeys corresponding to a secret key
sage: key = [1, 0, 1, 0, 0, 0, 0, 0, 1, 0]
sage: sdes.subkey(key, n=1)  # the first subkey
[1, 0, 1, 0, 0, 1, 0, 0]
sage: key = [1, 0, 1, 0, 0, 0, 0, 0, 1, 0]
sage: sdes.subkey(key, n=2)  # the second subkey
[0, 1, 0, 0, 0, 0, 1, 1]
```

Illustrating the S-boxes of S-DES and the Feistel round function:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: #
sage: # the S-boxes of simplified DES
sage: sdes = SimplifiedDES()
sage: sbox = sdes.sbox()
sage: sbox[0]; sbox[1]
(1, 0, 3, 2, 3, 2, 1, 0, 0, 2, 1, 3, 3, 1, 3, 2)
(0, 1, 2, 3, 2, 0, 1, 3, 3, 0, 1, 0, 2, 1, 0, 3)
sage: #
sage: # the Feistel round function
sage: B = [1, 0, 1, 1, 1, 1, 0, 1]
sage: K = [1, 1, 0, 1, 0, 1, 0, 1]
sage: sdes.permute_substitute(B, K)
[1, 0, 1, 0, 1, 1, 0, 1]
```

Chapter 6

Mini Advanced Encryption Standard

The Data Encryption Standard (DES) operates on 56-bit keys, giving rise to a key space of $2^{56} = 72,057,594,037,927,936$ possible keys. On 28th January 1997, RSA Laboratories launched a series of cryptographic challenges, one of which was to recover a plaintext that had been encrypted using DES. A few months after the challenge was launched, the DESCHALL (an abbreviation of DES Challenge) project led by Matt Curtin, Justin Dolske and Rocke Verser announced on 18th June 1997 that it had recovered the secret key required to decipher the ciphertext. Assisted by a world-wide network of people who donated their computers' spare time and organized via the Internet, DESCHALL attempted an exhaustive key search and succeeded in finding the key after searching nearly a quarter of the key space. The DESCHALL project not only demonstrated that a brute-force attack against DES was possible, but also that such an attack can be executed using commodity computer hardware. An account of the DESCHALL project can be found in Curtin [32], written by one of the leaders of the project.

In response to the success of the DESCHALL project, and theoretical attacks against DES such as the one presented by Campbell and Wiener [25], the National Institute of Standards and Technology (NIST) issued in 1997 a call for proposals of a new encryption standard. On 26th November 2001 after about five years of intense scrutiny of all submitted candidate algorithms, the Rijndael cipher [33] was officially adopted as the Advanced Encryption Standard (AES) and standardized in FIPS 197 [3].

The parameters of the full AES algorithm make the algorithm itself unsuitable for stepping through by hand. This chapter describes a simplified version of AES designed by Phan [93] and named Mini-AES. Phan's Mini-AES is a simplified variant of AES with all parameters significantly reduced, but preserving the general structure of AES to allow cryptology students to manually work through the simplified version. As an AES variant, Mini-AES is different from SR [27, 28], which is a family of parameterizable variants of AES designed as a framework for comparing different cryptanalytic techniques that can be brought to bear on the full AES algorithm. The central goal of Mini-AES is as a teaching tool in the same way that S-DES [102] and the simplified AES of Musa et al. [72] can be used in cryptology pedagogy.

Section 6.1 presents an algebraic structure known as a finite field and outlines a number of results concerning finite fields. This sets the stage for a discussion in section 6.2 of the finite field that is specific to Mini-AES. Building on the mathematical foundation outlined in the previous two sections, in section 6.3 we present the individual functions whose composition in various order constitute the Mini-AES encryption and decryption algorithms as discussed in section 6.4. The specification of Mini-AES as described in Phan [93] and the present chapter has been implemented as part of the Sage [111] standard library. The reference manual of our implementation is contained in Appendix D and full source code is available with the latest stable release of Sage, which as of this writing is Sage version 4.2.1. Finally, section 6.5 provides examples illustrating functionalities of our Sage implementation.

6.1 Structure of finite fields

Arithmetic in finite fields and the structure of such objects lay the mathematical foundation of AES and its many variants. This section provides a survey of algebraic techniques used in AES, Mini-AES, simplified AES and SR. Our discussion touches upon only enough algebraic concepts to allow for a description of Mini-AES. See Hungerford [49] for an in-depth coverage of algebra, or Lidl and Niederreiter [58] for a thorough survey of finite fields.

We begin with a definition of fields.

Definition 6.1. ***Finite field.*** *A* field *is a non-empty set F having two operations (usually denoted as addition and multiplication) such that the following properties hold. For all $a, b, c \in F$ we have*

1. *Closure for addition and multiplication: $a + b \in F$ and $ab \in F$.*
2. *Associative addition and multiplication: $a + (b + c) = (a + b) + c$ and $a(bc) = (ab)c$.*
3. *Commutative addition: $a + b = b + a$.*
4. *Commutative multiplication: $ab = ba$.*
5. *Additive identity: there is an element $0_F \in F$ such that $a + 0_F = a = 0_F + a$ for all $a \in F$.*
6. *Multiplicative identity: there is an element $1_F \in F$ such that $1_F \neq 0_F$ and $a1_F = a = 1_F a$ for all $a \in F$.*
7. *Distributive laws: $a(b + c) = ab + ac$ and $(a + b)c = ac + bc$.*
8. *Additive inverse: for each $a \in F$, the equations $a + x = 0_F$ and $x + a = 0_F$ have solutions in F.*
9. *Multiplicative inverse: for each $a \neq 0_F$ in F, the equations $ax = 1_F$ and $xa = 1_F$ have solutions in F.*

A field is said to be finite, hence the name finite field, if it contains a finite number of elements. The order *of a finite field is its number of elements.*

A non-empty set having two operations that satisfy properties 1, 2, 3, 5, 7 and 8 of Definition 6.1 is said to be a *ring*. The ring $\mathbf{Z}/n\mathbf{Z}$, for $n \in \mathbf{Z}$ such that $n > 1$, features prominently in the encryption and decryption processes of the shift cryptosystem (Chapter 3), the affine cryptosystem (Chapter 4), and simplified DES (Chapter 5). Let $I = \{0, 1, 2, \ldots, n-1\}$ consists of all non-negative integers less than n. Each congruence class in $\mathbf{Z}/n\mathbf{Z}$ can be labelled with a unique element in I, i.e. we have a bijection of sets between $\mathbf{Z}/n\mathbf{Z}$ and I. We use this bijection to carry the ring structure of $\mathbf{Z}/n\mathbf{Z}$ over to I. In effect, one can identify $\mathbf{Z}/n\mathbf{Z}$ as I and define the former as $\mathbf{Z}/n\mathbf{Z} = \{0, 1, 2, \ldots, n-1\}$. For each prime $p \in \mathbf{Z}$, it can be shown that $\mathbf{Z}/p\mathbf{Z}$ is a field of p elements (see Theorem 1.38, p.14 in [58]), hence $\mathbf{Z}/p\mathbf{Z}$ can be identified as the Galois field $\mathbf{F}_p$ of order p.

Denote by $\mathbf{F}_p[x]$ the set consisting of all polynomials with coefficients in the field $\mathbf{F}_p$. Let $f(x)$ be a polynomial in $\mathbf{F}_p[x]$ and let $\mathbf{F}_p[x]/(f(x))$ be the set of all congruence classes modulo $f(x)$. Whereas each prime p allows $\mathbf{Z}/p\mathbf{Z}$ to be a field, irreducible polynomials play a similar role for $\mathbf{F}_p[x]/(f(x))$. To see why this is the case, we require some definitions.

Definition 6.2. ***Irreducible polynomial.*** *Let F be a field and $F[x]$ a polynomial ring with coefficients in F. If $f(x), g(x) \in F[x]$, we say that $f(x)$ is an* associate *of $g(x)$ if there exists a non-zero $c \in F$ such that $f(x) = c \cdot g(x)$. A non-constant polynomial $p(x) \in F[x]$ is said to be* irreducible *if its only divisors are its associates and all the non-zero constant polynomials in $F[x]$.*

It can be shown (see Theorem 5.10, p.129 in [49]) that $\mathbf{F}_p[x]/(f(x))$ is a field if and only if $f(x)$ is an irreducible polynomial in $\mathbf{F}_p[x]$. Furthermore, elements of $\mathbf{F}_p[x]/(f(x))$ are polynomials in $\mathbf{F}_p[x]$ of degree at most $d-1$, with d being the degree of $f(x)$ (see Corollary 5.5, p.121 in [49]). Any two finite fields with the same order are isomorphic (see Corollary 10.27, p.367 in [49]).

6.2 The Mini-AES irreducible polynomial

Mini-AES considers the polynomial ring $\mathbf{F}_2[x]$ and the specific polynomial $f(x) = x^4 + x^3 + 1 \in \mathbf{F}_2[x]$, which is irreducible in $\mathbf{F}_2[x]$. The encryption and decryption algorithms operate on polynomials in the finite field

$$\mathbf{F}_2[x]/(x^4 + x^3 + 1) \tag{6.1}$$

of $2^4 = 16$ elements, all of which are enumerated in Table 6.1. The field in (6.1) is equivalent up to isomorphism to the finite field $\mathbf{F}_{2^4}$. In fact, all monic irreducible polynomial in $\mathbf{F}_2[x]$ of degree at most 4 are

$$\begin{aligned} f_1(x) &= x^4 + x^3 + x^2 + x + 1 \\ f_2(x) &= x^4 + x^3 + 1 \\ f_3(x) &= x^4 + x + 1 \end{aligned}$$

and any $f_i(x)$ results in a finite field $\mathbf{F}_2[x]/(f_i(x))$ that is isomorphic to (6.1).

It can be shown that there is a bijection from (6.1) to the Cartesian product $\mathbf{F}_2^4 = \mathbf{F}_2 \times \mathbf{F}_2 \times \mathbf{F}_2 \times \mathbf{F}_2$, allowing for the identification of each element of the finite field (6.1) as a 4-tuple (a_0, a_1, a_2, a_3) where each $a_i \in \mathbf{F}_2$. For ease of notation, we

write the 4-tuple as $a_0a_1a_2a_3$, which can be considered as a 4-bit string or a nibble. The field operation + on elements of (6.1) can be identified with the exclusive-or operator $\oplus$ on bits and bytes. The nibble $a_0a_1a_2a_3$ can in turn be considered as the binary representation of a non-negative integer. The conversion between nibbles, polynomials in (6.1), and integers is presented in Table 6.2.

0	x^3
1	$x^3 + 1$
x	$x^3 + x$
$x + 1$	$x^3 + x + 1$
x^2	$x^3 + x^2$
$x^2 + 1$	$x^3 + x^2 + 1$
$x^2 + x$	$x^3 + x^2 + x$
$x^2 + x + 1$	$x^3 + x^2 + x + 1$

Table 6.1: All 16 elements in the finite field $\mathbf{F}_2[x]/(x^4 + x^3 + 1)$.

nibble	Mini-AES polynomial	integer	nibble	Mini-AES polynomial	integer
0000	0	0	1000	x^3	8
0001	1	1	1001	$x^3 + 1$	9
0010	x	2	1010	$x^3 + x$	10
0011	$x + 1$	3	1011	$x^3 + x + 1$	11
0100	x^2	4	1100	$x^3 + x^2$	12
0101	$x^2 + 1$	5	1101	$x^3 + x^2 + 1$	13
0110	$x^2 + x$	6	1110	$x^3 + x^2 + x$	14
0111	$x^2 + x + 1$	7	1111	$x^3 + x^2 + x + 1$	15

Table 6.2: Converting between nibbles, Mini-AES polynomials and integers.

6.3 Components of Mini-AES

The Mini-AES encryption and decryption algorithms operate on 16-bit blocks of ciphertext/plaintext and a 16-bit secret key from which two round keys are derived using a key schedule. It follows that the key space consists of $2^{16} = 65,536$ possible keys. A message to be encrypted or decrypted must first be broken up into blocks of 16 bits each. The blocks can then be encrypted or decrypted individually one after the other, or each block can be encrypted/decrypted in parallel with the other blocks. Consider the set

$$\mathcal{M} = \mathcal{M}_2\big(\mathbf{F}_2[x]/(x^4 + x^3 + 1)\big) \tag{6.2}$$

of all 2×2 matrices with entries over $\mathbf{F}_2[x]/(x^4 + x^3 + 1)$. If $b = (b_0, b_1, b_2, b_3)$ is a 16-bit input block, each b_i is a nibble so that b_i can be considered as a polynomial in the finite field (6.1). Use Table 6.2 to obtain the specific polynomial corresponding to each b_i. The whole 16-bit block b is then structured as an element in the matrix space $\mathcal{M}$.

We have implemented Mini-AES as the class

`sage.crypto.block_cipher.miniaes.MiniAES`

in the Sage standard library. For the reference manual of our implementation, refer to Appendix D.

Before presenting the Mini-AES encryption and decryption algorithms, we first describe the individual components that make up those algorithms. Mini-AES is comprised of four individual components: NibbleSub, ShiftRow, MixColumn and KeyAddition. Each round of Mini-AES encryption or decryption involves applying some combination of these four components.

6.3.1 NibbleSub

The NibbleSub function

$$\gamma : \mathcal{M} \longrightarrow \mathcal{M}$$

takes as input a 2×2 matrix of nibbles and substitutes each of these nibbles according to the S-box given in Table 6.3. The values in the S-box of Table 6.3 are taken from the first row of the first S-box of DES. Using the conversion rule in Table 6.2, the NibbleSub S-box can equivalently be presented with entries in the finite field $\mathbf{F}_2[x]/(x^4 + x^3 + 1)$ (see Table 6.4).

Input	Output	Input	Output
0000	1110	1000	0011
0001	0100	1001	1010
0010	1101	1010	0110
0011	0001	1011	1100
0100	0010	1100	0101
0101	1111	1101	1001
0110	1011	1110	0000
0111	1000	1111	0111

Table 6.3: The S-box of NibbleSub.

However, note that the NibbleSub S-box presented in Table 6.3 (and equivalently in Table 6.4) is to be used in the encryption algorithm. The S-box for decryption is obtained from Table 6.3 by reversing the role of the "Input" and "Output" columns. Thus the previous "Input" column for encryption now becomes the "Output" column for decryption, and the previous "Output" column for encryption is now the "Input" column for decryption. The S-box used for decryption can then be specified as given in Table 6.5. Again, we can apply the conversion Table 6.2 to present the finite field elements in the NibbleSub decryption S-box as nibbles. Where the NibbleSub function is used in the decryption algorithm, we denote the NibbleSub decryption function as γ^{-1}.

NibbleSub is implemented in the method

`sage.crypto.block_cipher.miniaes.MiniAES.nibble_sub`

and its reference manual can be found in section D.2.12.

Input	Output
0	$x^3 + x^2 + x$
1	x^2
x	$x^3 + x^2 + 1$
$x + 1$	1
x^2	x
$x^2 + 1$	$x^3 + x^2 + x + 1$
$x^2 + x$	$x^3 + x + 1$
$x^2 + x + 1$	x^3
x^3	$x + 1$
$x^3 + 1$	$x^3 + x$
$x^3 + x$	$x^2 + x$
$x^3 + x + 1$	$x^3 + x^2$
$x^3 + x^2$	$x^2 + 1$
$x^3 + x^2 + 1$	$x^3 + 1$
$x^3 + x^2 + x$	0
$x^3 + x^2 + x + 1$	$x^2 + x + 1$

Table 6.4: Representing the NibbleSub S-box as elements of $\mathbf{F}_2[x]/(x^4 + x^3 + 1)$.

Input	Output
0	$x^3 + x^2 + x$
1	$x + 1$
x	x^2
$x + 1$	x^3
x^2	1
$x^2 + 1$	$x^3 + x^2$
$x^2 + x$	$x^3 + x$
$x^2 + x + 1$	$x^3 + x^2 + x + 1$
x^3	$x^2 + x + 1$
$x^3 + 1$	$x^3 + x^2 + 1$
$x^3 + x$	$x^3 + 1$
$x^3 + x + 1$	$x^2 + x$
$x^3 + x^2$	$x^3 + x + 1$
$x^3 + x^2 + 1$	x
$x^3 + x^2 + x$	0
$x^3 + x^2 + x + 1$	$x^2 + 1$

Table 6.5: The NibbleSub S-box for decryption.

6.3.2 ShiftRow

Similarly to NibbleSub, the ShiftRow function

$$\pi : \mathcal{M} \longrightarrow \mathcal{M}$$

also takes as input a 2×2 matrix of nibbles. However, instead of performing a substitution on the nibbles, π performs a rotation on each row of the input matrix. The first or zero-th row is left unchanged, while the second or row one is rotated left by one nibble. This has the effect of only interchanging the nibbles in the second row. Let b_0, b_1, b_2, b_3 be four nibbles arranged as the following 2×2 matrix

$$B = \begin{bmatrix} b_0 & b_2 \\ b_1 & b_3 \end{bmatrix}.$$

Then the application of π on B is the mapping

$$\begin{bmatrix} b_0 & b_2 \\ b_1 & b_3 \end{bmatrix} \longmapsto \begin{bmatrix} b_0 & b_2 \\ b_3 & b_1 \end{bmatrix}.$$

Note that rotating a row to the left by one nibble is equivalent to rotating the same row by one nibble to the right. Hence, the function π is its own inverse, i.e. $\pi^{-1} = \pi$.

ShiftRow is implemented in the method

`sage.crypto.block_cipher.miniaes.MiniAES.shift_row`

and its reference manual is given in section D.2.16.

6.3.3 MixColumn

Consider the matrix

$$A = \begin{bmatrix} x+1 & x \\ x & x+1 \end{bmatrix}$$

with entries over $\mathbf{F}_2[x]/(x^4 + x^3 + 1)$. Note that A is its own inverse matrix, i.e. $A = A^{-1}$ and thus the matrix products $A^{-1}A$ and AA^{-1} result in the 2×2 identity matrix. The MixColumn function

$$\theta : \mathcal{M} \longrightarrow \mathcal{M}$$

takes a 2×2 matrix of nibbles

$$C = \begin{bmatrix} c_0 & c_2 \\ c_1 & c_3 \end{bmatrix}.$$

Using Table 6.2, each nibble c_i is then converted to an equivalent element c_i' in $\mathbf{F}_2[x]/(x^4+x^3+1)$ to obtain the matrix C'. We then multiply A by C' according to the rules of matrix multiplication. Thus the application of θ on C is the mapping

$$\begin{bmatrix} c_0 & c_2 \\ c_1 & c_3 \end{bmatrix} \longmapsto \begin{bmatrix} x+1 & x \\ x & x+1 \end{bmatrix} \begin{bmatrix} c_0' & c_2' \\ c_1' & c_3' \end{bmatrix}.$$

To recover C, we multiply A^{-1} by AC' (on the left) to get $A^{-1}(AC') = (A^{-1}A)C' = C'$. Again, use Table 6.2 to convert elements of C' to their corresponding nibbles.

Therefore, the function θ is its own inverse. We have implemented MixColumn in the method

`sage.crypto.block_cipher.miniaes.MiniAES.mix_column`

of the Sage standard library. See section D.2.11 for its reference manual.

6.3.4 KeyAddition

The KeyAddition function

$$\sigma_{K_i} : \mathcal{M} \longrightarrow \mathcal{M}$$

relies on a round key K_i derived from the secret key K using a key schedule. Before describing the operation of σ_{K_i}, we first show how each round key is derived from the secret key.

Phan's Mini-AES is defined to have two rounds. The round key K_0 is generated and used prior to the first round, with round keys K_1 and K_2 being used in rounds 1 and 2 respectively. In total, there are three round keys, each generated from the secret key K.

Let $K = (k_0, k_1, k_2, k_3)$ be a 16-bit secret key and hence each k_i is a sub-block of 4 nibbles. Similarly, denote the 16-bit round keys as $K_0 = (w_0, w_1, w_2, w_3)$, $K_1 = (w_4, w_5, w_6, w_7)$ and $K_2 = (w_8, w_9, w_{10}, w_{11})$. Each of the two rounds of Mini-AES encryption or decryption has a corresponding round constant κ_i. The two round constants are defined as the nibbles $\kappa_1 = 0001$ and $\kappa_2 = 0010$. Equivalently, we can apply the conversion rules in Table 6.2 to express these round constants as elements in the finite field $\mathbf{F}_2[x]/(x^4 + x^3 + 1)$:

$$\kappa_1 = 1 \qquad \text{and} \qquad \kappa_2 = x.$$

The zero-th round key K_0 is defined to be the same as the secret key, i.e. $K_0 = K$ or equivalently $(k_0, k_1, k_2, k_3) = (w_0, w_1, w_2, w_3)$. The generation of the round keys K_1 and K_2 are presented in Table 6.6.

Round i	Round key K_i
1	$w_4 = w_0 + \gamma(w_3) + \kappa_1$
	$w_5 = w_1 + w_4$
	$w_6 = w_2 + w_5$
	$w_7 = w_3 + w_6$
2	$w_8 = w_4 + \gamma(w_7) + \kappa_2$
	$w_9 = w_5 + w_8$
	$w_{10} = w_6 + w_9$
	$w_{11} = w_7 + w_{10}$

Table 6.6: Generating the round keys of Mini-AES.

We are now ready to define the function σ_{K_i}. Let $D = (d_0, d_1, d_2, d_3)$ be a 16-bit block and express D as the 2×2 matrix

$$D = \begin{bmatrix} d_0 & d_2 \\ d_1 & d_3 \end{bmatrix}$$

over the matrix space (6.2). Similarly, let $K_i = (k_0, k_1, k_2, k_3)$ be the i-th round key for $i = 0, 1, 2$ and express K_i as a 2×2 matrix over the matrix space (6.2):

$$K_i = \begin{bmatrix} k_0 & k_2 \\ k_1 & k_3 \end{bmatrix}.$$

Then the KeyAddition function σ_{K_i} is given by the following matrix addition map:

$$\begin{bmatrix} d_0 & d_2 \\ d_1 & d_3 \end{bmatrix} \longmapsto \begin{bmatrix} d_0 & d_2 \\ d_1 & d_3 \end{bmatrix} + \begin{bmatrix} k_0 & k_2 \\ k_1 & k_3 \end{bmatrix}.$$

To recover D, we add $D + K_i$ to K_i to get $(D + K_i) + K_i$. As each matrix in $\mathcal{M}$ is its own additive inverse, it follows that $(D + K_i) + K_i = D + (K_i + K_i) = D$. Therefore, the function σ_{K_i} is its own additive inverse.

KeyAddition is implemented in the method

`sage.crypto.block_cipher.miniaes.MiniAES.add_key`

of the Sage standard library. Its reference manual is contained in section D.2.1.

6.4 Encryption and decryption functions

Having defined the component functions NibbleSub γ, ShiftRow π, MixColumn θ, and KeyAddition σ_{K_i}, we are now ready to define the encryption function in terms of these four components. The Mini-AES encryption function

$$\mathcal{E} : \mathcal{M} \times \mathcal{M} \longrightarrow \mathcal{M}$$

takes as input a 16-bit plaintext block P and a 16-bit secret key K to produce a 16-bit ciphertext block C. In terms of the four component functions, $\mathcal{E}$ can be expressed as the function composition

$$\mathcal{E} = \sigma_{K_2} \circ \pi \circ \gamma \circ \sigma_{K_1} \circ \theta \circ \pi \circ \gamma \circ \sigma_{K_0}$$

where the order of execution is from right to left.

Each of the two rounds that constitute Mini-AES encryption is a composition of various component functions. Prior to the first round, KeyAddition is run with round key K_0. The first round is a composition of all four component functions with round key K_1. The second round follows the same sequence of function composition as the first round, but using round key K_2 and excluding MixColumn. The rounds of Mini-AES encryption can then be expressed as

$$\mathcal{E} = \underbrace{\sigma_{K_2} \circ \pi \circ \gamma}_{\text{round 2}} \circ \underbrace{\sigma_{K_1} \circ \theta \circ \pi \circ \gamma}_{\text{round 1}} \circ \sigma_{K_0}.$$

The decryption function

$$\mathcal{D} : \mathcal{M} \times \mathcal{M} \longrightarrow \mathcal{M}$$

is the inverse of $\mathcal{E}$. From our discussion of the inverse component functions in section 6.3, the decryption function $\mathcal{D}$ can be written as

$$\begin{aligned}\mathcal{D} &= (\sigma_{K_2} \circ \pi \circ \gamma \circ \sigma_{K_1} \circ \theta \circ \pi \circ \gamma \circ \sigma_{K_0})^{-1} \\ &= \sigma_{K_0}^{-1} \circ \gamma^{-1} \circ \pi^{-1} \circ \theta^{-1} \circ \sigma_{K_1}^{-1} \circ \gamma^{-1} \circ \pi^{-1} \circ \sigma_{K_2}^{-1} \\ &= \sigma_{K_0} \circ \underbrace{\gamma^{-1} \circ \pi \circ \theta \circ \sigma_{K_1}}_{\text{round 2}} \circ \underbrace{\gamma^{-1} \circ \pi \circ \sigma_{K_2}}_{\text{round 1}}.\end{aligned}$$

The encryption and decryption functions of Mini-AES are illustrated in Figures 6.1 and 6.2, respectively, and implemented in the methods

```
sage.crypto.block_cipher.miniaes.MiniAES.encrypt
sage.crypto.block_cipher.miniaes.MiniAES.decrypt
```

Their reference manual is presented in sections D.2.5 and D.2.6, respectively.

6.5 Example Sage usage

This section provides some examples to illustrate functionalities of our Sage implementation of Mini-AES. Refer to Appendix D for the full reference manual of our implementation.

We can encrypt a plaintext and decrypt the result as follows:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: P = MS([K("x^3 + x"), K("x^2 + 1"), K("x^2 + x"), K("x^3 + x^2")]); P
<BLANKLINE>
[  x^3 + x   x^2 + 1]
[  x^2 + x x^3 + x^2]
sage: key = MS([K("x^3 + x^2"), K("x^3 + x"), K("x^3 + x^2 + x"), K("x^2 + x + 1")]); key
<BLANKLINE>
[    x^3 + x^2       x^3 + x]
[x^3 + x^2 + x   x^2 + x + 1]
sage: C = maes.encrypt(P, key); C
<BLANKLINE>
[            x       x^2 + x]
[x^3 + x^2 + x       x^3 + x]
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt; P
<BLANKLINE>
[  x^3 + x   x^2 + 1]
[  x^2 + x x^3 + x^2]
<BLANKLINE>
[  x^3 + x   x^2 + 1]
[  x^2 + x x^3 + x^2]
sage: plaintxt == P
True
```

Instead of working with elements over $\mathbf{F}_{2^4}[x]/(x^4+x^3+1)$, we can also work directly with binary strings:

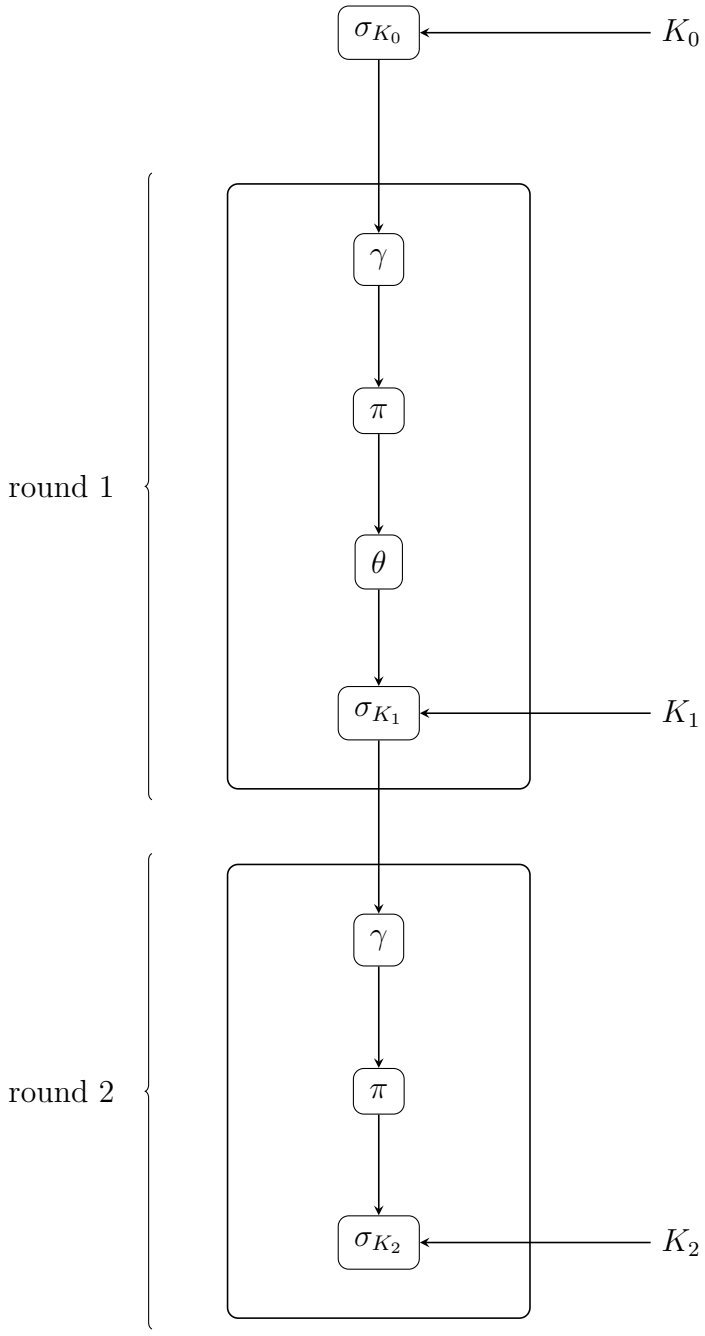

Figure 6.1: Two rounds in Mini-AES encryption.

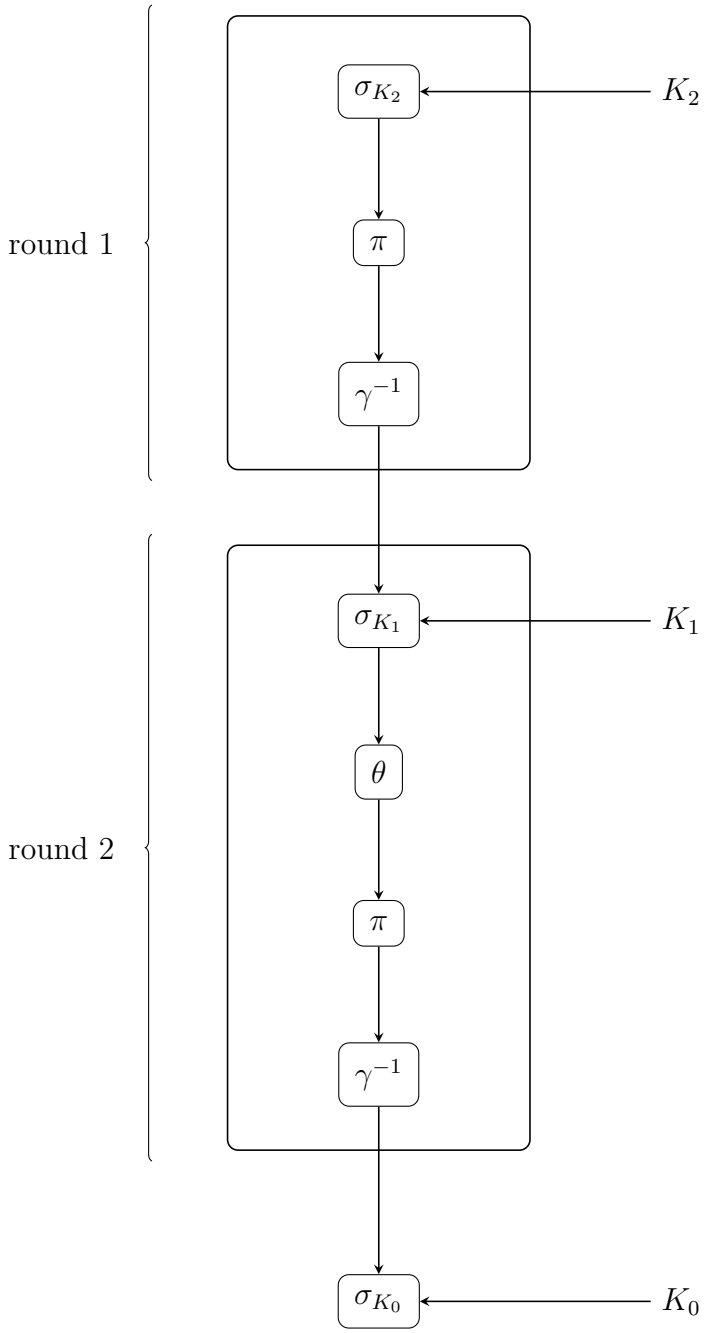

Figure 6.2: Two rounds in Mini-AES decryption.

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: bin = BinaryStrings()
sage: key = bin.encoding("KE"); key
0100101101000101
sage: P = bin.encoding("AB"); P
0100000101000010
sage: C = maes(P, key, algorithm="encrypt"); C
0101001100011011
sage: plaintxt = maes(C, key, algorithm="decrypt"Now we work with
sage: plaintxt == P
True
```

Or we could work with integers n such that $0 \leq n \leq 15$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: P = [n for n in xrange(16)]; P
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sage: key = [2, 3, 11, 0]; key
[2, 3, 11, 0]
sage: P = maes.integer_to_binary(P); P
0000000100100011010001010110011110001001101010111100110111101111
sage: key = maes.integer_to_binary(key); key
0010001110110000
sage: C = maes(P, key, algorithm="encrypt"); C
1100100000100011111001010101010101011011100111111000100001110000 1
sage: plaintxt = maes(C, key, algorithm="decrypt")
sage: plaintxt == P
True
```

Generate some random plaintext and a random secret key. Encrypt the plaintext using that secret key and decrypt the result. Then compare the decrypted plaintext with the original plaintext:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: MS = MatrixSpace(FiniteField(16, "x"), 2, 2)
sage: P = MS.random_element()
sage: key = maes.random_key()
sage: C = maes.encrypt(P, key)
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt == P
True
```

Obtaining the round keys from the secret key:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: key = MS([ [K("x^3 + x^2"), K("x^3 + x^2 + x + 1")], [K("x + 1"), K("0")] ])
sage: maes.round_key(key, 0)
<BLANKLINE>
[        x^3 + x^2 x^3 + x^2 + x + 1]
[            x + 1                 0]
sage: key
<BLANKLINE>
[        x^3 + x^2 x^3 + x^2 + x + 1]
[            x + 1                 0]
```

```
sage: maes.round_key(key, 1)
<BLANKLINE>
[            x + 1 x^3 + x^2 + x + 1]
[                0 x^3 + x^2 + x + 1]
sage: maes.round_key(key, 2)
<BLANKLINE>
[x^2 + x x^3 + 1]
[x^2 + x x^2 + x]
```

Here, we illustrate the operation of the function KeyAddition:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: D = MS([ [K("x^3 + x^2 + x + 1"), K("x^3 + x")], [K("0"), K("x^3 + x^2")] ]); D
<BLANKLINE>
[x^3 + x^2 + x + 1           x^3 + x]
[                0         x^3 + x^2]
sage: k = MS([ [K("x^2 + 1"), K("x^3 + x^2 + x + 1")], [K("x + 1"), K("0")] ]); k
<BLANKLINE>
[          x^2 + 1 x^3 + x^2 + x + 1]
[            x + 1                 0]
sage: maes.add_key(D, k)
<BLANKLINE>
[  x^3 + x   x^2 + 1]
[    x + 1 x^3 + x^2]
```

Illustrating the operation of the function MixColumn:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: mat = MS([ [K("x^2 + x + 1"), K("x^3 + x^2 + 1")], [K("x^3"), K("x")] ])
sage: maes.mix_column(mat)
<BLANKLINE>
[          x^3 + x                 0]
[          x^2 + 1 x^3 + x^2 + x + 1]
```

Illustrating the operation of the function NibbleSub:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: mat = MS([[K("x^3 + x^2 + x + 1"), K("0")], [K("x^2 + x + 1"), K("x^3 + x")]])
sage: maes.nibble_sub(mat, algorithm="encrypt")
<BLANKLINE>
[  x^2 + x + 1 x^3 + x^2 + x]
[          x^3       x^2 + x]
```

Illustrating the operation of the function ShiftRow:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: mat = MS([[K("x^3 + x^2 + x + 1"), K("0")], [K("x^2 + x + 1"), K("x^3 + x")]])
sage: maes.shift_row(mat)
<BLANKLINE>
[x^3 + x^2 + x + 1                0]
[         x^3 + x       x^2 + x + 1]
sage: mat
<BLANKLINE>
[x^3 + x^2 + x + 1                0]
[      x^2 + x + 1           x^3 + x]
```

Chapter 7

Conclusions and Future Work

We have surveyed the general purpose computer algebra systems (CAS) FriCAS, Maple, Mathematica, Matlab, Maxima and Sage with respect to their support for functionalities required for cryptography pedagogy. Based on this survey, we identified various strengths and weaknesses of each CAS and saw that Sage had more extensive support for cryptography education than the other CASs under consideration. Rather than setting out to match the other CASs feature-for-feature with Sage, we instead chose to build upon the existing extensive cryptographic functionalities in Sage in order to fill in various functionalities that our survey shows to be missing in Sage. The results are software implementation of the following cryptosystems:

- the shift cryptosystem, see Chapter 3 and Appendix A
- the affine cryptosystem, see Chapter 4 and Appendix B
- a simplified variant of the Data Encryption Standard called S-DES, see Chapter 5 and Appendix C
- and a simplified variant of the Advanced Encryption Standard called Mini-AES, see Chapter 6 and Appendix D.

We have also provided an implementation of an algorithm for solving the subset sum problem in the particular case of super-increasing sequences (see Appendix E), which serves as a foundation for future work on implementing knapsack cryptosystems. All of our enhancements to Sage described in the thesis have been accepted by the Sage development community and our software patches have been merged into the code base that constitutes the Sage standard library. The source code of our implementation is available with the latest stable release of Sage, which as of this writing is Sage 4.2.1.

Despite what has been accomplished in the thesis, much more work needs to be done before Sage has built-in support for all of the cryptographic functionalities identified by our survey in Chapter 2. A direction for future work would include building upon our support for knapsack problems based on super-increasing sequences to implement various knapsack cryptosystems. One could also implement wrapper code around the PyCrypto library to expose its functionalities for teaching the RSA, Rabin and ElGamal public-key cryptosystems, and their corresponding digital signature schemes including the Digital Signature Standard. Public-key cryptography based on number theoretic techniques are known to be less efficient in terms of speed

than symmetric-key cryptosystems. For cryptography pedagogy, an overriding concern is software support to enable student exploration of a particular cryptosystem. Hence, the time efficiency of an implementation is not an issue. However, one still needs to carefully choose algorithms to maintain a balance between a working implementation and an efficient working implementation. Our implementation of the shift and affine cryptosystems also provides some support for cryptanalyzing those two cryptosystems. Continuing along that line, one could implement techniques for cryptanalyzing all of the cryptosystems identified by our survey.

Once all of the cryptographic functionalities identified in Chapter 2 have been implemented, an ambitious direction for future research would include using the Sage implementation in a classroom setting. On the one hand, this would reduce the amount of duplicate implementation effort. On the other hand, student feedback could be used to enhance Sage's support for cryptographic functionalities.

Appendix A

Sage Manual for Shift Cryptosystem

The shift cryptosystem described in Chapter 3 is implemented in the class

`sage.crypto.classical.ShiftCryptosystem` (A.1)

via bug tracking tickets `#6841` [84], `#7010` [82], and `#7123` [78]. In this appendix, we provide the reference manual for the Sage class (A.1). The three bug tracking tickets `#6841`, `#7010` and `#7123` have been merged in the Sage standard library during the development of Sage versions 4.1.2.alpha4, 4.2.alpha0, and 4.2.alpha1 respectively. The source code of the class (A.1) is available with the latest release of Sage, which as of this writing is Sage version 4.2.1.

A.1 Class documentation

Create a shift cryptosystem.

Let $\mathcal{A} = \{a_0, a_1, a_2, \ldots, a_{n-1}\}$ be a non-empty alphabet consisting of n unique elements. Define a mapping $f : \mathcal{A} \longrightarrow \mathbf{Z}/n\mathbf{Z}$ from the alphabet $\mathcal{A}$ to the set $\mathbf{Z}/n\mathbf{Z}$ of integers modulo n, given by $f(a_i) = i$. Thus we can identify each element of the alphabet $\mathcal{A}$ with a unique integer $0 \leq i < n$. A key of the shift cipher is an integer $0 \leq k < n$. Therefore the key space is $\mathbf{Z}/n\mathbf{Z}$. Since we assume that $\mathcal{A}$ does not have repeated elements, the mapping $f : \mathcal{A} \longrightarrow \mathbf{Z}/n\mathbf{Z}$ is bijective. Encryption works by moving along the alphabet by k positions, with wrap around. Decryption reverses the process by moving backwards by k positions, with wrap around. More generally, let k be a secret key, i.e. an element of the key space, and let p be a plaintext character and consequently $p \in \mathbf{Z}/n\mathbf{Z}$. Then the ciphertext character c corresponding to p is given by

$$c \equiv p + k \pmod{n}.$$

Similarly, given a ciphertext character $c \in \mathbf{Z}/n\mathbf{Z}$ and a secret key k, we can recover the corresponding plaintext character as follows:

$$p \equiv c - k \pmod{n}.$$

Use the bijection $f : \mathcal{A} \longrightarrow \mathbf{Z}/n\mathbf{Z}$ to convert c and p back to elements of the alphabet $\mathcal{A}$. Currently, the following alphabets are supported for the shift cipher:

- Upper-case letters of the English alphabet as implemented in the function `AlphabeticStrings()`.
- The alphabet consisting of the hexadecimal number system as implemented in `HexadecimalStrings()`.
- The alphabet consisting of the binary number system as implemented in the function `BinaryStrings()`.

A.1.1 Example usage

Some examples illustrating encryption and decryption over various alphabets. Here is an example over the upper-case letters of the English alphabet:

```
sage: S = ShiftCryptosystem(AlphabeticStrings()); S
Shift cryptosystem on Free alphabetic string monoid on A-Z
sage: P = S.encoding("The shift cryptosystem generalizes the Caesar cipher.")
sage: P
THESHIFTCRYPTOSYSTEMGENERALIZESTHECAESARCIPHER
sage: K = 7
sage: C = S.enciphering(K, P); C
AOLZOPMAJYFWAVZFZALTNLULYHSPGLZAOLJHLZHYJPWOLY
sage: S.deciphering(K, C)
THESHIFTCRYPTOSYSTEMGENERALIZESTHECAESARCIPHER
sage: S.deciphering(K, C) == P
True
```

The previous example can also be done as follows:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("The shift cryptosystem generalizes the Caesar cipher.")
sage: K = 7
sage: E = S(K); E
Shift cipher on Free alphabetic string monoid on A-Z
sage: C = E(P); C
AOLZOPMAJYFWAVZFZALTNLULYHSPGLZAOLJHLZHYJPWOLY
sage: D = S(S.inverse_key(K)); D
Shift cipher on Free alphabetic string monoid on A-Z
sage: D(C) == P
True
sage: D(C) == P == D(E(P))
True
```

Over the hexadecimal number system:

```
sage: S = ShiftCryptosystem(HexadecimalStrings()); S
Shift cryptosystem on Free hexadecimal string monoid
sage: P = S.encoding("Shift encryption."); P
536869667420656e6372797074696f6e2e
sage: K = 5
sage: C = S.enciphering(K, P); C
a8bdbebbc975bab3b8c7cec5c9beb4b373
sage: S.deciphering(K, C)
536869667420656e6372797074696f6e2e
sage: S.deciphering(K, C) == P
True
```

And over the binary number system:

```
sage: S = ShiftCryptosystem(BinaryStrings()); S
Shift cryptosystem on Free binary string monoid
sage: P = S.encoding("Binary."); P
01000010011010010110111001100001011100100111100100101110
sage: K = 1
sage: C = S.enciphering(K, P); C
10111101100101101001000110011110100011011000011011010001
sage: S.deciphering(K, C)
01000010011010010110111001100001011100100111100100101110
sage: S.deciphering(K, C) == P
True
```

A shift cryptosystem with key $k = 3$ is commonly referred to as the Caesar cipher. Create a Caesar cipher over the upper-case letters of the English alphabet:

```
sage: caesar = ShiftCryptosystem(AlphabeticStrings())
sage: K = 3
sage: P = caesar.encoding("abcdef"); P
ABCDEF
sage: C = caesar.enciphering(K, P); C
DEFGHI
sage: caesar.deciphering(K, C) == P
True
```

Generate a random key for encryption and decryption:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Shift cipher with a random key.")
sage: K = S.random_key()
sage: C = S.enciphering(K, P)
sage: S.deciphering(K, C) == P
True
```

Decrypting with the key `K` is equivalent to encrypting with its corresponding inverse key:

```
sage: S.enciphering(S.inverse_key(K), C) == P
True
```

A.1.2 Exception tests

Currently, the octal number system is not supported as an alphabet for this shift cryptosystem:

```
sage: ShiftCryptosystem(OctalStrings())
...
TypeError: A (= Free octal string monoid) is not supported as a cipher\
domain of this shift cryptosystem.
```

Nor is the radix-64 number system supported:

```
sage: ShiftCryptosystem(Radix64Strings())
...
TypeError: A (= Free radix 64 string monoid) is not supported as a\
cipher domain of this shift cryptosystem.
```

Testing of dumping and loading objects:

```
sage: SA = ShiftCryptosystem(AlphabeticStrings())
sage: SA == loads(dumps(SA))
True
sage: SH = ShiftCryptosystem(HexadecimalStrings())
sage: SH == loads(dumps(SH))
True
sage: SB = ShiftCryptosystem(BinaryStrings())
sage: SB == loads(dumps(SB))
True
```

The key K must satisfy the inequality $0 \leq K < n$ with n being the size of the plaintext, ciphertext, and key spaces. For the shift cryptosystem, all these spaces are the same alphabet. This inequality must be satisfied for each of the supported alphabets. The capital letters of the English alphabet:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: S(2 + S.alphabet_size())
...
ValueError: K (=28) is outside the range of acceptable values for a\
key of this shift cryptosystem.
sage: S(-2)
...
ValueError: K (=-2) is outside the range of acceptable values for a\
key of this shift cryptosystem.
```

The hexadecimal number system:

```
sage: S = ShiftCryptosystem(HexadecimalStrings())
sage: S(1 + S.alphabet_size())
...
ValueError: K (=17) is outside the range of acceptable values for a\
key of this shift cryptosystem.
sage: S(-1)
...
ValueError: K (=-1) is outside the range of acceptable values for a\
key of this shift cryptosystem.
```

The binary number system:

```
sage: S = ShiftCryptosystem(BinaryStrings())
sage: S(1 + S.alphabet_size())
...
ValueError: K (=3) is outside the range of acceptable values for a key\
of this shift cryptosystem.
sage: S(-2)
...
ValueError: K (=-2) is outside the range of acceptable values for a\
key of this shift cryptosystem.
```

A.2 Public methods

This section documents public methods implemented in the class

`sage.crypto.classical.ShiftCryptosystem`

of the Sage standard library.

A.2.1 `brute_force(C, ranking='none')`

Attempt a brute force cryptanalysis of the ciphertext `C`.

Input

- `C` — A ciphertext over one of the supported alphabets of this shift cryptosystem. See the class `ShiftCryptosystem` for documentation on the supported alphabets.
- `ranking` — (default `"none"`) the method to use for ranking all possible keys. If `ranking="none"`, then do not use any ranking function. The following ranking functions are supported:
  - `"chisquare"` — the chi-square ranking function as implemented in the method `rank_by_chi_square()`.
  - `"squared_differences"` — the squared differences ranking function as implemented in the method `rank_by_squared_differences()`.

Output

- All the possible plaintext sequences corresponding to the ciphertext `C`. This method effectively uses all the possible keys in this shift cryptosystem to decrypt `C`. The method is also referred to as exhaustive key search. The output is a dictionary of key, plaintext pairs.

Examples

Cryptanalyze using all possible keys for various alphabets. Over the upper-case letters of the English alphabet:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("The shift cryptosystem generalizes the Caesar cipher.")
sage: K = 7
sage: C = S.enciphering(K, P)
sage: Dict = S.brute_force(C)
sage: for k in xrange(len(Dict)):
...       if Dict[k] == P:
...           print "key =", k
...
key = 7
```

Over the hexadecimal number system:

```
sage: S = ShiftCryptosystem(HexadecimalStrings())
sage: P = S.encoding("Encryption & decryption shifts along the alphabet.")
sage: K = 5
sage: C = S.enciphering(K, P)
sage: Dict = S.brute_force(C)
sage: for k in xrange(len(Dict)):
...       if Dict[k] == P:
...           print "key =", k
...
key = 5
```

And over the binary number system:

```
sage: S = ShiftCryptosystem(BinaryStrings())
sage: P = S.encoding("The binary alphabet is very insecure.")
sage: K = 1
sage: C = S.enciphering(K, P)
sage: Dict = S.brute_force(C)
sage: for k in xrange(len(Dict)):
...       if Dict[k] == P:
...           print "key =", k
...
key = 1
```

Don't use any ranking functions, i.e. `ranking="none"`:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Shifting using modular arithmetic.")
sage: K = 8
sage: C = S.enciphering(K, P)
sage: pdict = S.brute_force(C)
sage: sorted(pdict.items())
<BLANKLINE>
[(0, APQNBQVOCAQVOUWLCTIZIZQBPUMBQK),
(1, ZOPMAPUNBZPUNTVKBSHYHYPAOTLAPJ),
(2, YNOLZOTMAYOTMSUJARGXGXOZNSKZOI),
(3, XMNKYNSLZXNSLRTIZQFWFWNYMRJYNH),
(4, WLMJXMRKYWMRKQSHYPEVEVMXLQIXMG),
(5, VKLIWLQJXVLQJPRGXODUDULWKPHWLF),
(6, UJKHVKPIWUKPIOQFWNCTCTKVJOGVKE),
(7, TIJGUJOHVTJOHNPEVMBSBSJUINFUJD),
(8, SHIFTINGUSINGMODULARARITHMETIC),
(9, RGHESHMFTRHMFLNCTKZQZQHSGLDSHB),
(10, QFGDRGLESQGLEKMBSJYPYPGRFKCRGA),
(11, PEFCQFKDRPFKDJLARIXOXOFQEJBQFZ),
(12, ODEBPEJCQOEJCIKZQHWNWNEPDIAPEY),
(13, NCDAODIBPNDIBHJYPGVMVMDOCHZODX),
(14, MBCZNCHAOMCHAGIXOFULULCNBGYNCW),
(15, LABYMBGZNLBGZFHWNETKTKBMAFXMBV),
(16, KZAXLAFYMKAFYEGVMDSJSJALZEWLAU),
(17, JYZWKZEXLJZEXDFULCRIRIZKYDVKZT),
(18, IXYVJYDWKIYDWCETKBQHQHYJXCUJYS),
(19, HWXUIXCVJHXCVBDSJAPGPGXIWBTIXR),
(20, GVWTHWBUIGWBUACRIZOFOFWHVASHWQ),
(21, FUVSGVATHFVATZBQHYNENEVGUZRGVP),
(22, ETURFUZSGEUZSYAPGXMDMDUFTYQFUO),
(23, DSTQETYRFDTYRXZOFWLCLCTESXPETN),
(24, CRSPDSXQECSXQWYNEVKBKBSDRWODSM),
(25, BQROCRWPDBRWPVXMDUJAJARCQVNCRL)]
```

Use the chi-square ranking function, i.e. `ranking="chisquare"`:

```
sage: S.brute_force(C, ranking="chisquare")
<BLANKLINE>
[(8, SHIFTINGUSINGMODULARARITHMETIC),
(14, MBCZNCHAOMCHAGIXOFULULCNBGYNCW),
(20, GVWTHWBUIGWBUACRIZOFOFWHVASHWQ),
(13, NCDAODIBPNDIBHJYPGVMVMDOCHZODX),
(1, ZOPMAPUNBZPUNTVKBSHYHYPAOTLAPJ),
(23, DSTQETYRFDTYRXZOFWLCLCTESXPETN),
(10, QFGDRGLESQGLEKMBSJYPYPGRFKCRGA),
(6, UJKHVKPIWUKPIOQFWNCTCTKVJOGVKE),
(22, ETURFUZSGEUZSYAPGXMDMDUFTYQFUO),
(15, LABYMBGZNLBGZFHWNETKTKBMAFXMBV),
(12, ODEBPEJCQOEJCIKZQHWNWNEPDIAPEY),
(21, FUVSGVATHFVATZBQHYNENEVGUZRGVP),
(16, KZAXLAFYMKAFYEGVMDSJSJALZEWLAU),
(25, BQROCRWPDBRWPVXMDUJAJARCQVNCRL),
(9, RGHESHMFTRHMFLNCTKZQZQHSGLDSHB),
(24, CRSPDSXQECSXQWYNEVKBKBSDRWODSM),
(3, XMNKYNSLZXNSLRTIZQFWFWNYMRJYNH),
(5, VKLIWLQJXVLQJPRGXODUDULWKPHWLF),
(7, TIJGUJOHVTJOHNPEVMBSBSJUINFUJD),
(2, YNOLZOTMAYOTMSUJARGXGXOZNSKZOI),
(18, IXYVJYDWKIYDWCETKBQHQHYJXCUJYS),
(4, WLMJXMRKYWMRKQSHYPEVEVMXLQIXMG),
(11, PEFCQFKDRPFKDJLARIXOXOFQEJBQFZ),
(19, HWXUIXCVJHXCVBDSJAPGPGXIWBTIXR),
(0, APQNBQVOCAQVOUWLCTIZIZQBPUMBQK),
(17, JYZWKZEXLJZEXDFULCRIRIZKYDVKZT)]
```

Use the squared differences ranking function, i.e. `ranking="squared_differences"`:

```
sage: S.brute_force(C, ranking="squared_differences")
<BLANKLINE>
[(8, SHIFTINGUSINGMODULARARITHMETIC),
(23, DSTQETYRFDTYRXZOFWLCLCTESXPETN),
(12, ODEBPEJCQOEJCIKZQHWNWNEPDIAPEY),
(2, YNOLZOTMAYOTMSUJARGXGXOZNSKZOI),
(9, RGHESHMFTRHMFLNCTKZQZQHSGLDSHB),
(7, TIJGUJOHVTJOHNPEVMBSBSJUINFUJD),
(21, FUVSGVATHFVATZBQHYNENEVGUZRGVP),
(22, ETURFUZSGEUZSYAPGXMDMDUFTYQFUO),
(1, ZOPMAPUNBZPUNTVKBSHYHYPAOTLAPJ),
(16, KZAXLAFYMKAFYEGVMDSJSJALZEWLAU),
(20, GVWTHWBUIGWBUACRIZOFOFWHVASHWQ),
(24, CRSPDSXQECSXQWYNEVKBKBSDRWODSM),
(14, MBCZNCHAOMCHAGIXOFULULCNBGYNCW),
(13, NCDAODIBPNDIBHJYPGVMVMDOCHZODX),
(3, XMNKYNSLZXNSLRTIZQFWFWNYMRJYNH),
(10, QFGDRGLESQGLEKMBSJYPYPGRFKCRGA),
(15, LABYMBGZNLBGZFHWNETKTKBMAFXMBV),
(6, UJKHVKPIWUKPIOQFWNCTCTKVJOGVKE),
(11, PEFCQFKDRPFKDJLARIXOXOFQEJBQFZ),
(25, BQROCRWPDBRWPVXMDUJAJARCQVNCRL),
(17, JYZWKZEXLJZEXDFULCRIRIZKYDVKZT),
(19, HWXUIXCVJHXCVBDSJAPGPGXIWBTIXR),
(4, WLMJXMRKYWMRKQSHYPEVEVMXLQIXMG),
(0, APQNBQVOCAQVOUWLCTIZIZQBPUMBQK),
(18, IXYVJYDWKIYDWCETKBQHQHYJXCUJYS),
(5, VKLIWLQJXVLQJPRGXODUDULWKPHWLF)]
```

Exception tests

Currently, the octal number system is not supported as an alphabet for this shift cryptosystem:

```
sage: SA = ShiftCryptosystem(AlphabeticStrings())
sage: OctStr = OctalStrings()
sage: C = OctStr([1, 2, 3])
sage: SA.brute_force(C)
...
TypeError: ciphertext must be encoded using one of the supported\
cipher domains of this shift cryptosystem.
```

Nor is the radix-64 alphabet supported:

```
sage: Rad64 = Radix64Strings()
sage: C = Rad64([1, 2, 3])
sage: SA.brute_force(C)
...
TypeError: ciphertext must be encoded using one of the supported\
cipher domains of this shift cryptosystem.
```

A.2.2 `deciphering(K, C)`

Decrypt the ciphertext `C` with the key `K` using shift cipher decryption.

Input

- `K` — a secret key; a key belonging to the key space of this shift cipher. This key is an integer k satisfying the inequality $0 \leq k < n$, where n is the size of the cipher domain.
- `C` — a string of ciphertext; possibly an empty string. Characters in this string must be encoded using one of the supported alphabets. See the method `encoding()` for more information.

Output

- The plaintext corresponding to the ciphertext `C`.

Examples

Let's perform decryption over the supported alphabets. Here is decryption over the capital letters of the English alphabet:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Stop shifting me."); P
STOPSHIFTINGME
sage: K = 13
sage: C = S.enciphering(K, P); C
FGBCFUVSGVATZR
sage: S.deciphering(K, C) == P
True
```

Decryption over the hexadecimal number system:

```
sage: S = ShiftCryptosystem(HexadecimalStrings())
sage: P = S.encoding("Shift me now."); P
5368696674206d65206e6f772e
sage: K = 7
sage: C = S.enciphering(K, P); C
cadfd0ddeb97d4dc97d5d6ee95
sage: S.deciphering(K, C) == P
True
```

Decryption over the binary number system:

```
sage: S = ShiftCryptosystem(BinaryStrings())
sage: P = S.encoding("OK, enough shifting."); P
010011110100101100101100001000000110010101101110011011110111010101100111011\
010000010000001110011011010000110100101100110011101000110100101101110011001\
1100101110
sage: K = 1
sage: C = S.enciphering(K, P); C
101100001011010011010011110111111100110101001000110010000100010101001100010\
0
101111101111110001100100101111001011010011001100010111001011010010001100110\
0011010001
sage: S.deciphering(K, C) == P
True
```

A.2.3 `enciphering(K, P)`

Encrypt the plaintext `P` with the key `K` using shift cipher encryption.

Input

- `K` — a key belonging to the key space of this shift cipher. This key is an integer k satisfying the inequality $0 \leq k < n$, where n is the size of the cipher domain.
- `P` — a string of plaintext; possibly an empty string. Characters in this string must be encoded using one of the supported alphabets. See the method `encoding()` for more information.

Output

- The ciphertext corresponding to the plaintext `P`.

Examples

Let's perform encryption over the supported alphabets. Here is encryption over the capital letters of the English alphabet:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Shift your gear."); P
SHIFTYOURGEAR
sage: K = 3
sage: S.enciphering(K, P)
VKLIWBRXUJHDU
```

Encryption over the hexadecimal number system:

```
sage: S = ShiftCryptosystem(HexadecimalStrings())
sage: P = S.encoding("Capitalize with the shift key."); P
4361706974616c697a652077697468207468652073686967742 06b65792e
sage: K = 5
sage: S.enciphering(K, P)
98b6c5bec9b6b1becfba75ccbec9bd75c9bdba75c8bdbebbc975b0bace73
```

Encryption over the binary number system:

```
sage: S = ShiftCryptosystem(BinaryStrings())
sage: P = S.encoding("Don't␣shift.");␣P
01000100011011110110111000100111011101000010000001110011011010000110100101 1\
00110011101000010111 0
sage:␣K␣=␣1
sage:␣S.enciphering(K,␣P)
10111011100100001001000111011000100010111101111110001100100101111001011010 0\
11001100010111101000 1
```

A.2.4 `encoding(S)`

The encoding of the string `S` over the string monoid of this shift cipher. For example, if the string monoid of this cryptosystem is `AlphabeticStringMonoid`, then the encoding of `S` would be its upper-case equivalent stripped of all non-alphabetic characters. The following alphabets are supported for the shift cipher:

- Upper-case letters of the English alphabet as implemented in the function `AlphabeticStrings()`.
- The alphabet consisting of the hexadecimal number system as implemented in `HexadecimalStrings()`.
- The alphabet consisting of the binary number system as implemented in the function `BinaryStrings()`.

Input

- `S` — a string, possibly empty.

Output

- The encoding of `S` over the string monoid of this cryptosystem. If `S` is an empty string, return an empty string.

Examples

Encoding over the upper-case letters of the English alphabet:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: S.encoding("Shift cipher on capital letters of the English alphabet.")
SHIFTCIPHERONCAPITALLETTERSOFTHEENGLISHALPHABET
```

Encoding over the binary system:

```
sage: S = ShiftCryptosystem(BinaryStrings())
sage: S.encoding("Binary")
010000100110100101101110011000010111001001111001
```

Encoding over the hexadecimal system:

```
sage: S = ShiftCryptosystem(HexadecimalStrings())
sage: S.encoding("Over hexadecimal system.")
4f766572206865786164563696d616c2073797374656d2e
```

The argument `S` can be an empty string, in which case an empty string is returned:

```
sage: ShiftCryptosystem(AlphabeticStrings()).encoding("")
<BLANKLINE>
sage: ShiftCryptosystem(HexadecimalStrings()).encoding("")
<BLANKLINE>
sage: ShiftCryptosystem(BinaryStrings()).encoding("")
<BLANKLINE>
```

A.2.5 `inverse_key(K)`

The inverse key corresponding to the key `K`. For the shift cipher, the inverse key corresponding to `K` is $-K \bmod n$, where $n > 0$ is the size of the cipher domain, i.e. the plaintext/ciphertext space. A key k of the shift cipher is an integer $0 \leq k < n$. The key $k = 0$ has no effect on either the plaintext or the ciphertext.

Input

- `K` — a key for this shift cipher. This must be an integer k such that $0 \leq k < n$, where n is the size of the cipher domain.

Output

- The inverse key corresponding to `K`.

Examples

Some random keys and their respective inverse keys:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: key = S.random_key(); key  # random
2
sage: S.inverse_key(key)         # random
24
sage: S = ShiftCryptosystem(HexadecimalStrings())
sage: key = S.random_key(); key  # random
12
sage: S.inverse_key(key)         # random
4
sage: S = ShiftCryptosystem(BinaryStrings())
sage: key = S.random_key(); key  # random
1
sage: S.inverse_key(key)         # random
1
```

```
sage: key = S.random_key(); key  # random
0
sage: S.inverse_key(key)         # random
0
```

Regardless of the value of a key, the addition of the key and its inverse must be equal to the alphabet size. This relationship holds exactly when the value of the key is non-zero:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: K = S.random_key()
sage: while K == 0:
...       K = S.random_key()
...
sage: invK = S.inverse_key(K)
sage: K + invK == S.alphabet_size()
True
sage: invK + K == S.alphabet_size()
True
sage: K = S.random_key()
sage: while K != 0:
...       K = S.random_key()
...
sage: invK = S.inverse_key(K)
sage: K + invK != S.alphabet_size()
True
sage: K; invK
0
0
```

Exception tests

The key K must satisfy the inequality $0 \leq K < n$ with n being the size of the plaintext, ciphertext, and key spaces. For the shift cryptosystem, all these spaces are the same alphabet. This inequality must be satisfied for each of the supported alphabets. The capital letters of the English alphabet:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: S.inverse_key(S.alphabet_size())
...
ValueError: K (=26) is outside the range of acceptable values for a\
key of this shift cryptosystem.
sage: S.inverse_key(-1)
...
ValueError: K (=-1) is outside the range of acceptable values for a\
key of this shift cryptosystem.
```

The hexadecimal number system:

```
sage: S = ShiftCryptosystem(HexadecimalStrings())
sage: S.inverse_key(S.alphabet_size())
...
ValueError: K (=16) is outside the range of acceptable values for a\
key of this shift cryptosystem.
sage: S.inverse_key(-1)
...
ValueError: K (=-1) is outside the range of acceptable values for a\
key of this shift cryptosystem.
```

The binary number system:

```
sage: S = ShiftCryptosystem(BinaryStrings())
sage: S.inverse_key(S.alphabet_size())
...
ValueError: K (=2) is outside the range of acceptable values for a key\
of this shift cryptosystem.
sage: S.inverse_key(-1)
...
ValueError: K (=-1) is outside the range of acceptable values for a\
key of this shift cryptosystem.
```

A.2.6 random_key()

Generate a random key within the key space of this shift cipher. The generated key is an integer $0 \leq k < n$ with n being the size of the cipher domain. Thus there are n possible keys in the key space, which is the set $\mathbf{Z}/n\mathbf{Z}$. The key $k = 0$ has no effect on either the plaintext or the ciphertext.

Output

- A random key within the key space of this shift cryptosystem.

Examples

Generating random keys for shift cryptosystems over different alphabets:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: S.random_key()  # random
18
sage: S = ShiftCryptosystem(BinaryStrings())
sage: S.random_key()  # random
0
sage: S = ShiftCryptosystem(HexadecimalStrings())
sage: S.random_key()  # random
5
```

Regardless of the value of a key, the addition of the key and its inverse must be equal to the alphabet size. This relationship holds exactly when the value of the key is non-zero:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: K = S.random_key()
sage: while K == 0:
...       K = S.random_key()
...
sage: invK = S.inverse_key(K)
sage: K + invK == S.alphabet_size()
True
sage: invK + K == S.alphabet_size()
True
sage: K = S.random_key()
sage: while K != 0:
...       K = S.random_key()
...
sage: invK = S.inverse_key(K)
sage: K + invK != S.alphabet_size()
```

```
True
sage: K; invK
0
0
```

A.2.7 rank_by_chi_square(C, pdict)

Use the chi-square statistic to rank all possible keys. Currently, this method only applies to the capital letters of the English alphabet.

Algorithm

Consider a non-empty alphabet $\mathcal{A}$ consisting of n elements, and let C be a ciphertext encoded using elements of $\mathcal{A}$. The plaintext P corresponding to C is also encoded using elements of $\mathcal{A}$. Let M be a candidate decipherment of C, i.e. M is the result of attempting to decrypt C using a key $k \in \mathbf{Z}/n\mathbf{Z}$ which is not necessarily the same key used to encrypt P. Suppose $F_{\mathcal{A}}(e)$ is the characteristic frequency probability of $e \in \mathcal{A}$ and let $F_M(e)$ be the message frequency probability with respect to M. The characteristic frequency probability distribution of an alphabet is the expected frequency probability distribution for that alphabet. The message frequency probability distribution of M provides a distribution of the ratio of character occurrences over message length. One can interpret the characteristic frequency probability $F_{\mathcal{A}}(e)$ as the expected probability, while the message frequency probability $F_M(e)$ is the observed probability. If M is of length L, then the observed frequency of $e \in \mathcal{A}$ is

$$O_M(e) = F_M(e) \cdot L$$

and the expected frequency of $e \in \mathcal{A}$ is

$$E_{\mathcal{A}}(e) = F_{\mathcal{A}}(e) \cdot L.$$

The chi-square rank $R_{\chi^2}(M)$ of M corresponding to a key $k \in \mathbf{Z}/n\mathbf{Z}$ is given by

$$R_{\chi^2}(M) = \sum_{e \in \mathcal{A}} \frac{\big(O_M(e) - E_{\mathcal{A}}(e)\big)^2}{E_{\mathcal{A}}(e)}.$$

Cryptanalysis by exhaustive key search produces a candidate decipherment M_k for each possible key $k \in \mathbf{Z}/n\mathbf{Z}$. For a set $D = \{M_{k_1}, M_{k_2}, \ldots, M_{k_r}\}$ of all candidate decipherments corresponding to a ciphertext C, the smaller is the rank $R_{\chi^2}(M_{k_i})$ the more likely that k_i is the secret key. This key ranking method is based on the Pearson chi-square test.

Input

- `C` — The ciphertext, a non-empty string. The ciphertext must be encoded using the upper-case letters of the English alphabet.

- `pdict` — A dictionary of key, possible plaintext pairs. This should be the output of `brute_force()` with `ranking="none"`.

Output

- A list ranking the most likely keys first. Each element of the list is a tuple of key, possible plaintext pairs.

Examples

Use the chi-square statistic to rank all possible keys and their corresponding decipherment:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Shi."); P
SHI
sage: K = 5
sage: C = S.enciphering(K, P)
sage: Pdict = S.brute_force(C)
sage: S.rank_by_chi_square(C, Pdict)
<BLANKLINE>
[(9, ODE),
(5, SHI),
(20, DST),
(19, ETU),
(21, CRS),
(10, NCD),
(25, YNO),
(6, RGH),
(12, LAB),
(8, PEF),
(1, WLM),
(11, MBC),
(18, FUV),
(17, GVW),
(2, VKL),
(4, TIJ),
(3, UJK),
(0, XMN),
(16, HWX),
(15, IXY),
(23, APQ),
(24, ZOP),
(22, BQR),
(7, QFG),
(13, KZA),
(14, JYZ)]
```

As more ciphertext is available, the reliability of the chi-square ranking function increases:

```
sage: P = S.encoding("Shift cipher."); P
SHIFTCIPHER
sage: C = S.enciphering(K, P)
sage: Pdict = S.brute_force(C)
sage: S.rank_by_chi_square(C, Pdict)
<BLANKLINE>
[(5, SHIFTCIPHER),
(9, ODEBPYELDAN),
(18, FUVSGPVCURE),
(2, VKLIWFLSKHU),
(20, DSTQENTASPC),
(19, ETURFOUBTQD),
(21, CRSPDMSZROB),
(6, RGHESBHOGDQ),
(7, QFGDRAGNFCP),
```

```
(12, LABYMVBIAXK),
(17, GVWTHQWDVSF),
(24, ZOPMAJPWOLY),
(1, WLMJXGMTLIV),
(0, XMNKYHNUMJW),
(11, MBCZNWCJBYL),
(8, PEFCQZFMEBO),
(25, YNOLZIOVNKX),
(10, NCDAOXDKCZM),
(3, UJKHVEKRJGT),
(4, TIJGUDJQIFS),
(22, BQROCLRYQNA),
(16, HWXUIRXEWTG),
(15, IXYVJSYFXUH),
(14, JYZWKTZGYVI),
(13, KZAXLUAHZWJ),
(23, APQNBKQXPMZ)]
```

Exception tests

The ciphertext cannot be an empty string:

```
sage: S.rank_by_chi_square("", Pdict)
...
AttributeError: 'str' object has no attribute 'parent'
sage: S.rank_by_chi_square(S.encoding(""), Pdict)
...
ValueError: The ciphertext must be a non-empty string.
sage: S.rank_by_chi_square(S.encoding(" "), Pdict)
...
ValueError: The ciphertext must be a non-empty string.
```

The ciphertext must be encoded using the capital letters of the English alphabet as implemented in `AlphabeticStrings()`:

```
sage: H = HexadecimalStrings()
sage: S.rank_by_chi_square(H.encoding("shift"), Pdict)
...
TypeError: The ciphertext must be capital letters of the English
alphabet.
sage: B = BinaryStrings()
sage: S.rank_by_chi_square(B.encoding("shift"), Pdict)
...
TypeError: The ciphertext must be capital letters of the English
alphabet.
```

The dictionary `pdict` cannot be empty:

```
sage: S.rank_by_chi_square(C, {})
...
KeyError: 0
```

A.2.8 `rank_by_squared_differences(C, pdict)`

Use the squared-differences measure to rank all possible keys. Currently, this method only applies to the capital letters of the English alphabet.

Algorithm

Consider a non-empty alphabet $\mathcal{A}$ consisting of n elements, and let C be a ciphertext encoded using elements of $\mathcal{A}$. The plaintext P corresponding to C is also encoded using elements of $\mathcal{A}$. Let M be a candidate decipherment of C, i.e. M is the result of attempting to decrypt C using a key $k \in \mathbf{Z}/n\mathbf{Z}$ which is not necessarily the same key used to encrypt P. Suppose $F_{\mathcal{A}}(e)$ is the characteristic frequency probability of $e \in \mathcal{A}$ and let $F_M(e)$ be the message frequency probability with respect to M. The characteristic frequency probability distribution of an alphabet is the expected frequency probability distribution for that alphabet. The message frequency probability distribution of M provides a distribution of the ratio of character occurrences over message length. One can interpret the characteristic frequency probability $F_{\mathcal{A}}(e)$ as the expected probability, while the message frequency probability $F_M(e)$ is the observed probability. If M is of length L, then the observed frequency of $e \in \mathcal{A}$ is

$$O_M(e) = F_M(e) \cdot L$$

and the expected frequency of $e \in \mathcal{A}$ is

$$E_{\mathcal{A}}(e) = F_{\mathcal{A}}(e) \cdot L.$$

The squared-differences, or residual sum of squares, rank $R_{RSS}(M)$ of M corresponding to a key $k \in \mathbf{Z}/n\mathbf{Z}$ is given by

$$R_{RSS}(M) = \sum_{e \in \mathcal{A}} \big(O_M(e) - E_{\mathcal{A}}(e)\big)^2.$$

Cryptanalysis by exhaustive key search produces a candidate decipherment M_k for each possible key $k \in \mathbf{Z}/n\mathbf{Z}$. For a set $D = \{M_{k_1}, M_{k_2}, \ldots, M_{k_r}\}$ of all candidate decipherments corresponding to a ciphertext C, the smaller is the rank $R_{RSS}(M_{k_i})$ the more likely that k_i is the secret key. This key ranking method is based on the residual sum of squares measure.

Input

- `C` — The ciphertext, a non-empty string. The ciphertext must be encoded using the upper-case letters of the English alphabet.
- `pdict` — A dictionary of key, possible plaintext pairs. This should be the output of `brute_force()` with `ranking="none"`.

Output

- A list ranking the most likely keys first. Each element of the list is a tuple of key, possible plaintext pairs.

Examples

Use the method of squared differences to rank all possible keys and their corresponding decipherment:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Shi."); P
SHI
sage: K = 5
sage: C = S.enciphering(K, P)
sage: Pdict = S.brute_force(C)
sage: S.rank_by_squared_differences(C, Pdict)
<BLANKLINE>
[(19, ETU),
(9, ODE),
(20, DST),
(5, SHI),
(8, PEF),
(4, TIJ),
(25, YNO),
(21, CRS),
(6, RGH),
(10, NCD),
(12, LAB),
(23, APQ),
(24, ZOP),
(0, XMN),
(13, KZA),
(15, IXY),
(1, WLM),
(16, HWX),
(22, BQR),
(11, MBC),
(18, FUV),
(2, VKL),
(17, GVW),
(7, QFG),
(3, UJK),
(14, JYZ)]
```

As more ciphertext is available, the reliability of the squared differences ranking function increases:

```
sage: P = S.encoding("Shift cipher."); P
SHIFTCIPHER
sage: C = S.enciphering(K, P)
sage: Pdict = S.brute_force(C)
sage: S.rank_by_squared_differences(C, Pdict)
<BLANKLINE>
[(20, DSTQENTASPC),
(5, SHIFTCIPHER),
(9, ODEBPYELDAN),
(19, ETURFOUBTQD),
(6, RGHESBHOGDQ),
(16, HWXUIRXEWTG),
(8, PEFCQZFMEBO),
(21, CRSPDMSZROB),
(22, BQROCLRYQNA),
(25, YNOLZIOVNKX),
(3, UJKHVEKRJGT),
(18, FUVSGPVCURE),
(4, TIJGUDJQIFS),
(10, NCDAOXDKCZM),
(7, QFGDRAGNFCP),
(24, ZOPMAJPWOLY),
(2, VKLIWFLSKHU),
(12, LABYMVBIAXK),
(17, GVWTHQWDVSF),
(1, WLMJXGMTLIV),
(13, KZAXLUAHZWJ),
(0, XMNKYHNUMJW),
```

```
(15, IXYVJSYFXUH),
(14, JYZWKTZGYVI),
(11, MBCZNWCJBYL),
(23, APQNBKQXPMZ)]
```

Exception tests

The ciphertext cannot be an empty string:

```
sage: S.rank_by_squared_differences("", Pdict)
...
AttributeError: 'str' object has no attribute 'parent'
sage: S.rank_by_squared_differences(S.encoding(""), Pdict)
...
ValueError: The ciphertext must be a non-empty string.
sage: S.rank_by_squared_differences(S.encoding(" "), Pdict)
...
ValueError: The ciphertext must be a non-empty string.
```

The ciphertext must be encoded using the capital letters of the English alphabet as implemented in `AlphabeticStrings()`:

```
sage: H = HexadecimalStrings()
sage: S.rank_by_squared_differences(H.encoding("shift"), Pdict)
...
TypeError: The ciphertext must be capital letters of the English
alphabet.
sage: B = BinaryStrings()
sage: S.rank_by_squared_differences(B.encoding("shift"), Pdict)
...
TypeError: The ciphertext must be capital letters of the English
alphabet.
```

The dictionary `pdict` cannot be empty:

```
sage: S.rank_by_squared_differences(C, {})
...
KeyError: 0
```

A.3 Private methods

This section documents private methods implemented in the class

`sage.crypto.classical.ShiftCryptosystem`

of the Sage standard library.

A.3.1 `__init__(A)`

See `ShiftCryptosystem` for full documentation. Create a shift cryptosystem defined over the alphabet `A`.

Input

- `A` — a string monoid over some alphabet; this is the non-empty alphabet over which the plaintext and ciphertext spaces are defined.

Output

- A shift cryptosystem over the alphabet `A`.

Examples

```
sage: S = ShiftCryptosystem(AlphabeticStrings()); S
Shift cryptosystem on Free alphabetic string monoid on A-Z
sage: P = S.encoding("The shift cryptosystem generalizes the Caesar cipher.")
sage: P
THESHIFTCRYPTOSYSTEMGENERALIZESTHECAESARCIPHER
sage: K = 7
sage: C = S.enciphering(K, P); C
AOLZOPMAJYFWAVZFZALTNLULYHSPGLZAOLJHLZHYJPWOLY
sage: S.deciphering(K, C)
THESHIFTCRYPTOSYSTEMGENERALIZESTHECAESARCIPHER
sage: S.deciphering(K, C) == P
True
```

A.3.2 `__call__(K)`

Create a shift cipher with key `K`.

Input

- `K` — a secret key; this key is used for both encryption and decryption. For the shift cryptosystem whose plaintext and ciphertext spaces are $\mathcal{A}$, a key is any integer k such that $0 \leq k < n$ where n is the size or cardinality of the set $\mathcal{A}$.

Output

- A shift cipher with secret key `K`.

Examples

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: P = S.encoding("Shifting sand."); P
SHIFTINGSAND
sage: K = 3
sage: E = S(K); E
Shift cipher on Free alphabetic string monoid on A-Z
sage: E(P)
VKLIWLQJVDQG
sage: D = S(S.inverse_key(K)); D
Shift cipher on Free alphabetic string monoid on A-Z
sage: D(E(P))
SHIFTINGSAND
```

Exception tests

The key `K` must satisfy the inequality $0 \leq K < n$ with n being the size of the plaintext, ciphertext, and key spaces. For the shift cryptosystem, all these spaces are the same alphabet. This inequality must be satisfied for each of the supported alphabets. The capital letters of the English alphabet:

```
sage: S = ShiftCryptosystem(AlphabeticStrings())
sage: S(2 + S.alphabet_size())
Traceback (most recent call last):
...
ValueError: K (=28) is outside the range of acceptable values for a\
key of this shift cryptosystem.
sage: S(-2)
Traceback (most recent call last):
...
ValueError: K (=-2) is outside the range of acceptable values for a\
key of this shift cryptosystem.
```

The hexadecimal number system:

```
sage: S = ShiftCryptosystem(HexadecimalStrings())
sage: S(1 + S.alphabet_size())
Traceback (most recent call last):
...
ValueError: K (=17) is outside the range of acceptable values for a\
key of this shift cryptosystem.
sage: S(-1)
Traceback (most recent call last):
...
ValueError: K (=-1) is outside the range of acceptable values for a\
key of this shift cryptosystem.
```

The binary number system:

```
sage: S = ShiftCryptosystem(BinaryStrings())
sage: S(1 + S.alphabet_size())
Traceback (most recent call last):
...
ValueError: K (=3) is outside the range of acceptable values for a\
key of this shift cryptosystem.
sage: S(-2)
Traceback (most recent call last):
...
ValueError: K (=-2) is outside the range of acceptable values for a\
key of this shift cryptosystem.
```

A.3.3 `_repr_()`

Return the string representation of this shift cryptosystem.

Examples

```
sage: ShiftCryptosystem(AlphabeticStrings())
Shift cryptosystem on Free alphabetic string monoid on A-Z
sage: ShiftCryptosystem(HexadecimalStrings())
Shift cryptosystem on Free hexadecimal string monoid
sage: ShiftCryptosystem(BinaryStrings())
Shift cryptosystem on Free binary string monoid
```

Appendix B

Sage Manual for Affine Cryptosystem

The affine cryptosystem described in Chapter 4 is implemented in the class

`sage.crypto.classical.AffineCryptosystem` (B.1)

via bug tracking ticket `#7124` [76]. This appendix contains the reference manual for the class (B.1). Our implementation of the affine cryptosystem has been merged into the Sage standard library during the development of Sage 4.2.1.alpha0.

B.1 Class documentation

Create an affine cryptosystem.

Let $\mathcal{A} = \{a_0, a_1, a_2, \ldots, a_{n-1}\}$ be a non-empty alphabet consisting of n unique elements. Define a mapping $f : \mathcal{A} \longrightarrow \mathbf{Z}/n\mathbf{Z}$ from the alphabet $\mathcal{A}$ to the set $\mathbf{Z}/n\mathbf{Z}$ of integers modulo n, given by $f(a_i) = i$. Thus we can identify each element of the alphabet $\mathcal{A}$ with a unique integer $0 \leq i < n$. A key of the affine cipher is an ordered pair of integers $(a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ such that $\gcd(a, n) = 1$. Therefore the key space is $\mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$. Since we assume that $\mathcal{A}$ does not have repeated elements, the mapping $f : \mathcal{A} \longrightarrow \mathbf{Z}/n\mathbf{Z}$ is bijective. Encryption and decryption functions are both affine functions. Let (a, b) be a secret key, i.e. an element of the key space, and let p be a plaintext character and consequently $p \in \mathbf{Z}/n\mathbf{Z}$. Then the ciphertext character c corresponding to p is given by

$$c \equiv ap + b \pmod{n}.$$

Similarly, given a ciphertext character $c \in \mathbf{Z}/n\mathbf{Z}$ and a secret key (a, b), we can recover the corresponding plaintext character as follows:

$$p \equiv a^{-1}(c - b) \pmod{n}$$

where a^{-1} is the inverse of a modulo n. Use the bijection $f : \mathcal{A} \longrightarrow \mathbf{Z}/n\mathbf{Z}$ to convert c and p back to elements of the alphabet $\mathcal{A}$. Currently, only the following alphabet is supported for the affine cipher:

- Upper-case letters of the English alphabet as implemented in the function `AlphabeticStrings()`.

B.1.1 Examples

Encryption and decryption over the capital letters of the English alphabet:

```
sage: A = AffineCryptosystem(AlphabeticStrings()); A
Affine cryptosystem on Free alphabetic string monoid on A-Z
sage: P = A.encoding("The affine cryptosystem generalizes the shift cipher.")
sage: P
THEAFFINECRYPTOSYSTEMGENERALIZESTHESHIFTCIPHER
sage: a, b = (9, 13)
sage: C = A.enciphering(a, b, P); C
CYXNGGHAXFKVSCJTVTCXRPXAXKNIHEXTCYXTYHGCFHSYXK
sage: A.deciphering(a, b, C)
THEAFFINECRYPTOSYSTEMGENERALIZESTHESHIFTCIPHER
sage: A.deciphering(a, b, C) == P
True
```

We can also use functional notation to work through the previous example:

```
sage: A = AffineCryptosystem(AlphabeticStrings()); A
Affine cryptosystem on Free alphabetic string monoid on A-Z
sage: P = A.encoding("The affine cryptosystem generalizes the shift cipher.")
sage: P
THEAFFINECRYPTOSYSTEMGENERALIZESTHESHIFTCIPHER
sage: a, b = (9, 13)
sage: E = A(a, b); E
Affine cipher on Free alphabetic string monoid on A-Z
sage: C = E(P); C
CYXNGGHAXFKVSCJTVTCXRPXAXKNIHEXTCYXTYHGCFHSYXK
sage: aInv, bInv = A.inverse_key(a, b)
sage: D = A(aInv, bInv); D
Affine cipher on Free alphabetic string monoid on A-Z
sage: D(C)
THEAFFINECRYPTOSYSTEMGENERALIZESTHESHIFTCIPHER
sage: D(C) == P
True
sage: D(C) == P == D(E(P))
True
```

Encrypting the ciphertext with the inverse key also produces the plaintext:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: P = A.encoding("Encrypt with inverse key.")
sage: a, b = (11, 8)
sage: C = A.enciphering(a, b, P)
sage: P; C
ENCRYPTWITHINVERSEKEY
AVENMRJQSJHSVFANYAOAM
sage: aInv, bInv = A.inverse_key(a, b)
sage: A.enciphering(aInv, bInv, C)
ENCRYPTWITHINVERSEKEY
sage: A.enciphering(aInv, bInv, C) == P
True
```

For a secret key $(a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$, if $a = 1$ then any affine cryptosystem with key $(1, b)$ for any $b \in \mathbf{Z}/n\mathbf{Z}$ is a shift cryptosystem. Here is how we can create a Caesar cipher using the affine cryptosystem:

```
sage: caesar = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (1, 3)
sage: P = caesar.encoding("abcdef"); P
ABCDEF
sage: C = caesar.enciphering(a, b, P); C
DEFGHI
sage: caesar.deciphering(a, b, C) == P
True
```

Any affine cipher with keys of the form $(a, 0) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ is called a decimation cipher on the Roman alphabet, or decimation cipher for short:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: P = A.encoding("A decimation cipher is a specialized affine cipher.")
sage: a, b = (17, 0)
sage: C = A.enciphering(a, b, P)
sage: P; C
ADECIMATIONCIPHERISASPECIALIZEDAFFINECIPHER
AZQIGWALGENIGVPQDGUAUVQIGAFGJQZAHHGNQIGVPQD
sage: A.deciphering(a, b, C) == P
True
```

Generate a random key for encryption and decryption:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: P = A.encoding("An affine cipher with a random key.")
sage: a, b = A.random_key()
sage: C = A.enciphering(a, b, P)
sage: A.deciphering(a, b, C) == P
True
```

B.1.2 Exception tests

The binary number system is currently not a supported alphabet of this affine cryptosystem:

```
sage: AffineCryptosystem(BinaryStrings())
...
TypeError: A (= Free binary string monoid) is not supported as a\
cipher domain of this affine cryptosystem.
```

Nor are the octal, hexadecimal, and radix-64 number systems supported:

```
sage: AffineCryptosystem(OctalStrings())
...
TypeError: A (= Free octal string monoid) is not supported as a cipher\
domain of this affine cryptosystem.
sage: AffineCryptosystem(HexadecimalStrings())
...
TypeError: A (= Free hexadecimal string monoid) is not supported as a\
cipher domain of this affine cryptosystem.
sage: AffineCryptosystem(Radix64Strings())
...
TypeError: A (= Free radix 64 string monoid) is not supported as a\
cipher domain of this affine cryptosystem.
```

A secret key (a, b) must be an element of $\mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ with $\gcd(a, n) = 1$. This rules out the case $a = 0$ irrespective of the value of b. For the upper-case letters of the English alphabet, where the alphabet size is $n = 26$, a cannot take on any even value:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: A(0, 1)
...
ValueError: (a, b) = (0, 1) is outside the range of acceptable values\
for a key of this affine cryptosystem.
sage: A(2, 1)
...
ValueError: (a, b) = (2, 1) is outside the range of acceptable values\
for a key of this affine cryptosystem.
```

B.2 Public methods

This section documents public methods implemented in the class

`sage.crypto.classical.AffineCryptosystem`

of the Sage standard library.

B.2.1 `brute_force(C, ranking='none')`

Attempt a brute force cryptanalysis of the ciphertext `C`.

Input

- `C` — A ciphertext over one of the supported alphabets of this affine cryptosystem. See the class `AffineCryptosystem` for documentation on the supported alphabets.

- `ranking` — (default `"none"`) the method to use for ranking all possible keys. If `ranking="none"`, then do not use any ranking function. The following ranking functions are supported:

  - `"chi_square"` — the chi-square ranking function as implemented in the method `rank_by_chi_square()`.
  - `"squared_differences"` — the squared differences ranking function as implemented in the method `rank_by_squared_differences()`.

Output

- All the possible plaintext sequences corresponding to the ciphertext `C`. This method effectively uses all the possible keys in this affine cryptosystem to decrypt `C`. The method is also referred to as exhaustive key search. The output is a dictionary of key, candidate decipherment pairs.

Examples

Cryptanalyze using all possible keys with the option `ranking="none"`:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (3, 7)
sage: P = A.encoding("Linear"); P
LINEAR
sage: C = A.enciphering(a, b, P)
sage: L = A.brute_force(C)
sage: sorted(L.items())[:26]  # display 26 candidate decipherments
<BLANKLINE>
[((1, 0), OFUTHG),
((1, 1), NETSGF),
((1, 2), MDSRFE),
((1, 3), LCRQED),
((1, 4), KBQPDC),
((1, 5), JAPOCB),
((1, 6), IZONBA),
((1, 7), HYNMAZ),
((1, 8), GXMLZY),
((1, 9), FWLKYX),
((1, 10), EVKJXW),
((1, 11), DUJIWV),
((1, 12), CTIHVU),
((1, 13), BSHGUT),
((1, 14), ARGFTS),
((1, 15), ZQFESR),
((1, 16), YPEDRQ),
((1, 17), XODCQP),
((1, 18), WNCBPO),
((1, 19), VMBAON),
((1, 20), ULAZNM),
((1, 21), TKZYML),
((1, 22), SJYXLK),
((1, 23), RIXWKJ),
((1, 24), QHWVJI),
((1, 25), PGVUIH)]
```

Use the chi-square ranking function, i.e. `ranking="chisquare"`:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (3, 7)
sage: P = A.encoding("Linear functions for encrypting and decrypting."); P
LINEARFUNCTIONSFORENCRYPTINGANDDECRYPTING
sage: C = A.enciphering(a, b, P)
sage: Rank = A.brute_force(C, ranking="chisquare")
sage: Rank[:10]  # display only the top 10 candidate keys
<BLANKLINE>
[((3, 7), LINEARFUNCTIONSFORENCRYPTINGANDDECRYPTING),
((23, 25), VYTCGPBMTENYSTOBSPCTEPIRNYTAGTDDCEPIRNYTA),
((1, 12), CTIHVUKDIBATLIXKLUHIBUPOATINVIEEHBUPOATIN),
((11, 15), HSRYELDAROVSWRQDWLYROLUBVSRIERTTYOLUBVSRI),
((25, 1), NWHIUVFMHOPWEHSFEVIHOVABPWHCUHLLIOVABPWHC),
((25, 7), TCNOABLSNUVCKNYLKBONUBGHVCNIANRROUBGHVCNI),
((15, 4), SHIBVOWZILEHDIJWDOBILOFYEHIRVIGGBLOFYEHIR),
((15, 23), PEFYSLTWFIBEAFGTALYFILCVBEFOSFDDYILCVBEFO),
((7, 10), IDUFHSYXUTEDNULYNSFUTSVGEDURHUMMFTSVGEDUR),
((19, 22), QVETRGABEFUVLENALGTEFGDSUVEHREMMTFGDSUVEH)]
```

Use the squared differences ranking function, i.e. `ranking="squared_differences"`:

```
sage: Rank = A.brute_force(C, ranking="squared_differences")
sage: Rank[:10]  # display only the top 10 candidate keys
<BLANKLINE>
[((3, 7), LINEARFUNCTIONSFORENCRYPTINGANDDECRYPTING),
((23, 6), GJENRAMXEPYJDEZMDANEPATCYJELREOONPATCYJEL),
((23, 25), VYTCGPBMTENYSTOBSPCTEPIRNYTAGTDDCEPIRNYTA),
((19, 22), QVETRGABEFUVLENALGTEFGDSUVEHREMMTFGDSUVEH),
((19, 9), DIRGETNORSHIYRANYTGRSTQFHIRUERZZGSTQFHIRU),
((23, 18), KNIRVEQBITCNHIDQHERITEXGCNIPVISSRTEXGCNIP),
((17, 16), GHORBEIDOJMHFOVIFEROJETWMHOZBOAARJETWMHOZ),
((21, 14), AHEZRMOFEVQHTEBOTMZEVMNIQHEDREKKZVMNIQHED),
((1, 12), CTIHVUKDIBATLIXKLUHIBUPOATINVIEEHBUPOATIN),
((7, 18), SNEPRCIHEDONXEVIXCPEDCFQONEBREWWPDCFQONEB)]
```

Exception tests

Currently, the binary number system is not supported as an alphabet of this affine cryptosystem:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: BinStr = BinaryStrings()
sage: C = BinStr.encoding("abc")
sage: A.brute_force(C)
...
TypeError: Ciphertext must be encoded using one of the supported\
cipher domains of this affine cryptosystem.
```

Nor are the octal, hexadecimal, and radix-64 number systems supported:

```
sage: OctStr = OctalStrings()
sage: C = OctStr([1, 2, 3])
sage: A.brute_force(C)
...
TypeError: Ciphertext must be encoded using one of the supported\
cipher domains of this affine cryptosystem.
sage: HexStr = HexadecimalStrings()
sage: C = HexStr.encoding("abc")
sage: A.brute_force(C)
...
TypeError: Ciphertext must be encoded using one of the supported\
cipher domains of this affine cryptosystem.
sage: RadStr = Radix64Strings()
sage: C = RadStr([1, 2, 3])
sage: A.brute_force(C)
...
TypeError: Ciphertext must be encoded using one of the supported\
cipher domains of this affine cryptosystem.
```

Only the chi-square and squared-differences ranking functions are currently supported. The keyword ranking must take on either of the values `"none"`, `"chisquare"` or `"squared_differences"`:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (3, 7)
sage: P = A.encoding("Linear")
sage: C = A.enciphering(a, b, P)
sage: A.brute_force(C, ranking="chi")
...
ValueError: Keyword 'ranking' must be either 'none', 'chisquare', or\
```

```
'squared_differences'.
sage: A.brute_force(C, ranking="")
...
ValueError: Keyword 'ranking' must be either 'none', 'chisquare', or\
'squared_differences'.
```

B.2.2 deciphering(a, b, C)

Decrypt the ciphertext `C` with the key `(a, b)` using affine cipher decryption.

Input

- `a, b` — a secret key belonging to the key space of this affine cipher. This key must be an element of $\mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ such that $\gcd(a, n) = 1$ with n being the size of the ciphertext and plaintext spaces.
- `C` — a string of ciphertext; possibly an empty string. Characters in this string must be encoded using one of the supported alphabets. See the method `encoding()` for more information.

Output

- The plaintext corresponding to the ciphertext `C`.

Examples

Decryption over the capital letters of the English alphabet:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (5, 2)
sage: P = A.encoding("Affine functions are linear functions.")
sage: C = A.enciphering(a, b, P); C
CBBQPWBYPMTQUPOCJWFQPWCJBYPMTQUPO
sage: P == A.deciphering(a, b, C)
True
```

The previous example can also be worked through using functional notation:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (5, 2)
sage: P = A.encoding("Affine functions are linear functions.")
sage: E = A(a, b); E
Affine cipher on Free alphabetic string monoid on A-Z
sage: C = E(P); C
CBBQPWBYPMTQUPOCJWFQPWCJBYPMTQUPO
sage: aInv, bInv = A.inverse_key(a, b)
sage: D = A(aInv, bInv); D
Affine cipher on Free alphabetic string monoid on A-Z
sage: D(C) == P
True
```

If the ciphertext is an empty string, then the plaintext is also an empty string regardless of the value of the secret key:

```
sage: a, b = A.random_key()
sage: A.deciphering(a, b, A.encoding(""))
<BLANKLINE>
sage: A.deciphering(a, b, A.encoding(" "))
<BLANKLINE>
```

Exception tests

The key must be an ordered pair $(a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ with n being the size of the plaintext and ciphertext spaces. Furthermore, a must be relatively prime to n, i.e. $\gcd(a, n) = 1$:

```
sage: A.deciphering(2, 6, P)
...
ValueError: (a, b) = (2, 6) is outside the range of acceptable values\
for a key of this affine cipher.
```

B.2.3 `enciphering(a, b, P)`

Encrypt the plaintext `P` with the key `(a, b)` using affine cipher encryption.

Input

- `a, b` — a secret key belonging to the key space of this affine cipher. This key must be an element of $\mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ such that $\gcd(a, n) = 1$ with n being the size of the ciphertext and plaintext spaces.
- `P` — a string of plaintext; possibly an empty string. Characters in this string must be encoded using one of the supported alphabets. See the method `encoding()` for more information.

Output

- The ciphertext corresponding to the plaintext `P`.

Examples

Encryption over the capital letters of the English alphabet:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (3, 6)
sage: P = A.encoding("Affine ciphers work with linear functions.")
sage: A.enciphering(a, b, P)
GVVETSMEZBSFIUWFKUELBNETSGFVOTMLEWTI
```

Now work through the previous example using functional notation:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (3, 6)
sage: P = A.encoding("Affine ciphers work with linear functions.")
sage: E = A(a, b); E
Affine cipher on Free alphabetic string monoid on A-Z
sage: E(P)
GVVETSMEZBSFIUWFKUELBNETSGFVOTMLEWTI
```

If the plaintext is an empty string, then the ciphertext is also an empty string regardless of the value of the secret key:

```
sage: a, b = A.random_key()
sage: A.enciphering(a, b, A.encoding(""))
<BLANKLINE>
sage: A.enciphering(a, b, A.encoding(" "))
<BLANKLINE>
```

Exception tests

The key must be an ordered pair $(a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ with n being the size of the plaintext and ciphertext spaces. Furthermore, a must be relatively prime to n, i.e. $\gcd(a, n) = 1$:

```
sage: A.enciphering(2, 6, P)
...
ValueError: (a, b) = (2, 6) is outside the range of acceptable values\
for a key of this affine cryptosystem.
```

B.2.4 `encoding(S)`

The encoding of the string `S` over the string monoid of this affine cipher. For example, if the string monoid of this cryptosystem is `AlphabeticStringMonoid`, then the encoding of `S` would be its upper-case equivalent stripped of all non-alphabetic characters. Only the following alphabet is supported for the affine cipher:

- Upper-case letters of the English alphabet as implemented in the function `AlphabeticStrings()`.

Input

- `S` — a string, possibly empty.

Output

- The encoding of `S` over the string monoid of this cryptosystem. If `S` is an empty string, return an empty string.

Examples

Encoding over the upper-case letters of the English alphabet:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: A.encoding("Affine cipher over capital letters of the English alphabet.")
AFFINECIPHEROVERCAPITALLETTERSOFTHEENGLISHALPHABET
```

The argument `S` can be an empty string, in which case an empty string is returned:

```
sage: AffineCryptosystem(AlphabeticStrings()).encoding("")
<BLANKLINE>
```

B.2.5 `inverse_key(a, b)`

The inverse key corresponding to the secret key `(a,b)`. If p is a plaintext character so that $p \in \mathbf{Z}/n\mathbf{Z}$ and n is the alphabet size, then the ciphertext c corresponding to p is

$$c \equiv ap + b \pmod{n}.$$

As (a, b) is a key, then the multiplicative inverse a^{-1} exists and the original plaintext can be recovered as follows

$$\begin{aligned} p &\equiv a^{-1}(c - b) \pmod{n} \\ &\equiv a^{-1}c + a^{-1}(-b) \pmod{n}. \end{aligned}$$

Therefore the ordered pair $(a^{-1}, -ba^{-1})$ is the inverse key corresponding to (a, b).

Input

- `a, b` — a secret key for this affine cipher. The ordered pair `(a,b)` must be an element of $\mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ such that $\gcd(a, n) = 1$.

Output

- The inverse key $(a^{-1}, -ba^{-1})$ corresponding to (a, b).

Examples

Generating inverse keys of various keys:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (1, 2)
sage: A.inverse_key(a, b)
(1, 24)
sage: A.inverse_key(3, 2)
(9, 8)
```

Suppose that the plaintext and ciphertext spaces are the capital letters of the English alphabet so that $n = 26$. If $\varphi(n)$ is the Euler phi function of n, then there are $\varphi(n)$ integers $0 \leq a < n$ that are relatively prime to n. For the capital letters of the English alphabet, there are 12 such integers relatively prime to n:

```
sage: euler_phi(A.alphabet_size())
12
```

And here is a list of those integers:

```
sage: n = A.alphabet_size()
sage: L = [i for i in xrange(n) if gcd(i, n) == 1]; L
[1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25]
```

Then a secret key (a, b) of this shift cryptosystem is such that a is an element of the list `L` in the last example. Any inverse key (A, B) corresponding to (a, b) is such that A is also in the list `L` above:

```
sage: a, b = (3, 9)
sage: a in L
True
sage: aInv, bInv = A.inverse_key(a, b)
sage: aInv, bInv
(9, 23)
sage: aInv in L
True
```

Exception tests

Any ordered pair of the form $(0, b)$ for any integer b cannot be a secret key of this affine cipher. Hence $(0, b)$ does not have a corresponding inverse key:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: A.inverse_key(0, 1)
...
ValueError: (a, b) = (0, 1) is outside the range of acceptable values\
for a key of this affine cipher.
```

B.2.6 `random_key()`

Generate a random key within the key space of this affine cipher. The generated secret key is an ordered pair $(a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ with n being the size of the cipher domain and $\gcd(a, n) = 1$. Let $\varphi(n)$ denote the Euler phi function of n. Then the affine cipher has $n \cdot \varphi(n)$ possible keys.

Output

- A random key within the key space of this affine cryptosystem. The output key is an ordered pair (a, b).

Examples

Generating a random key:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: A.random_key()  # random
(17, 25)
```

If (a, b) is a secret key and n is the size of the plaintext and ciphertext alphabets, then $\gcd(a, n) = 1$:

```
sage: a, b = A.random_key()
sage: n = A.alphabet_size()
sage: gcd(a, n)
1
```

B.2.7 `rank_by_chi_square(C, pdict)`

Use the chi-square statistic to rank all possible keys. Currently, this method only applies to the capital letters of the English alphabet.

Algorithm

Consider a non-empty alphabet $\mathcal{A}$ consisting of n elements, and let C be a ciphertext encoded using elements of $\mathcal{A}$. The plaintext P corresponding to C is also encoded using elements of $\mathcal{A}$. Let M be a candidate decipherment of C, i.e. M is the result of attempting to decrypt C using a key (a, b) which is not necessarily the same key used to encrypt P. Suppose $F_{\mathcal{A}}(e)$ is the characteristic frequency probability of $e \in \mathcal{A}$ and let $F_M(e)$ be the message frequency probability with respect to M. The characteristic frequency probability distribution of an alphabet is the expected frequency probability distribution for that alphabet. The message frequency probability distribution of M provides a distribution of the ratio of character occurrences over message length. One can interpret the characteristic frequency probability $F_{\mathcal{A}}(e)$ as the expected probability, while the message frequency probability $F_M(e)$ is the observed probability. If M is of length L, then the observed frequency of $e \in \mathcal{A}$ is

$$O_M(e) = F_M(e) \cdot L$$

and the expected frequency of $e \in \mathcal{A}$ is

$$E_{\mathcal{A}}(e) = F_{\mathcal{A}}(e) \cdot L.$$

The chi-square rank $R_{\chi^2}(M)$ of M corresponding to a key $(a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ is given by

$$R_{\chi^2}(M) = \sum_{e \in \mathcal{A}} \frac{\big(O_M(e) - E_{\mathcal{A}}(e)\big)^2}{E_{\mathcal{A}}(e)}.$$

Cryptanalysis by exhaustive key search produces a candidate decipherment $M_{a,b}$ for each possible key (a, b). For a set $D = \{M_{a_1,b_1}, M_{a_2,b_2}, \ldots, M_{a_k,b_k}\}$ of all candidate decipherments corresponding to a ciphertext C, the smaller is the rank $R_{\chi^2}(M_{a_i,b_i})$ the more likely that (a_i, b_i) is the secret key. This key ranking method is based on the Pearson chi-square test.

Input

- `C` — The ciphertext, a non-empty string. The ciphertext must be encoded using the upper-case letters of the English alphabet.

- `pdict` — A dictionary of key, possible plaintext pairs. This should be the output of `brute_force()` with `ranking="none"`.

Output

- A list ranking the most likely keys first. Each element of the list is a tuple of key, possible plaintext pairs.

Examples

Use the chi-square statistic to rank all possible keys and their corresponding decipherment:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (3, 7)
sage: P = A.encoding("Line.")
sage: C = A.enciphering(a, b, P)
sage: Plist = A.brute_force(C)
sage: Rank = A.rank_by_chi_square(C, Plist)
sage: Rank[:10]  # display only the top 10 candidate keys
<BLANKLINE>
[((1, 1), NETS),
((3, 7), LINE),
((17, 20), STAD),
((5, 2), SLOT),
((5, 5), HADI),
((9, 25), TSLI),
((17, 15), DELO),
((15, 6), ETUN),
((21, 8), ELID),
((7, 17), HCTE)]
```

As more ciphertext is available, the reliability of the chi-square ranking function increases:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (11, 24)
sage: P = A.encoding("Longer message is more information for cryptanalysis.")
sage: C = A.enciphering(a, b, P)
sage: Plist = A.brute_force(C)
sage: Rank = A.rank_by_chi_square(C, Plist)
sage: Rank[:10]  # display only the top 10 candidate keys
<BLANKLINE>
[((11, 24), LONGERMESSAGEISMOREINFORMATIONFORCRYPTANALYSIS),
((17, 9), INURFSBFLLHRFDLBNSFDUYNSBHEDNUYNSTSVGEHUHIVLDL),
((9, 18), RMFIUHYUOOSIUWOYMHUWFBMHYSVWMFBMHGHETVSFSREOWO),
((15, 12), VSTACPUCOOGACYOUSPCYTBSPUGNYSTBSPEPIRNGTGVIOYO),
((3, 22), PAFOYLKYGGSOYEGKALYEFTALKSBEAFTALILCVBSFSPCGEG),
((25, 3), OHSRNADNPPFRNVPDHANVSCHADFEVHSCHAJABWEFSFOBPVP),
((7, 25), GHYNVIPVRRLNVFRPHIVFYEHIPLAFHYEHIDITQALYLGTRFR),
((5, 2), NEHCIVKISSUCIWSKEVIWHFEVKUPWEHFEVOVABPUHUNASWS),
((15, 25), IFGNPCHPBBTNPLBHFCPLGOFCHTALFGOFCRCVEATGTIVBLB),
((9, 6), BWPSERIEYYCSEGYIWREGPLWRICFGWPLWRQRODFCPCBOYGY)]
```

Exception tests

The ciphertext cannot be an empty string:

```
sage: A.rank_by_chi_square("", Plist)
...
AttributeError: 'str' object has no attribute 'parent'
sage: A.rank_by_chi_square(A.encoding(""), Plist)
...
ValueError: The ciphertext must be a non-empty string.
sage: A.rank_by_chi_square(A.encoding(" "), Plist)
...
ValueError: The ciphertext must be a non-empty string.
```

The ciphertext must be encoded using the capital letters of the English alphabet as implemented in `AlphabeticStrings()`:

```
sage: H = HexadecimalStrings()
sage: A.rank_by_chi_square(H.encoding("shift"), Plist)
...
TypeError: The ciphertext must be capital letters of the English alphabet.
sage: B = BinaryStrings()
sage: A.rank_by_chi_square(B.encoding("shift"), Plist)
...
TypeError: The ciphertext must be capital letters of the English alphabet.
```

The dictionary `pdict` cannot be empty:

```
sage: A.rank_by_chi_square(C, {})
...
KeyError: (1, 0)
```

B.2.8 `rank_by_squared_differences(C, pdict)`

Use the squared-differences measure to rank all possible keys. Currently, this method only applies to the capital letters of the English alphabet.

Algorithm

Consider a non-empty alphabet $\mathcal{A}$ consisting of n elements, and let C be a ciphertext encoded using elements of $\mathcal{A}$. The plaintext P corresponding to C is also encoded using elements of $\mathcal{A}$. Let M be a candidate decipherment of C, i.e. M is the result of attempting to decrypt C using a key (a, b) which is not necessarily the same key used to encrypt P. Suppose $F_{\mathcal{A}}(e)$ is the characteristic frequency probability of $e \in \mathcal{A}$ and let $F_M(e)$ be the message frequency probability with respect to M. The characteristic frequency probability distribution of an alphabet is the expected frequency probability distribution for that alphabet. The message frequency probability distribution of M provides a distribution of the ratio of character occurrences over message length. One can interpret the characteristic frequency probability $F_{\mathcal{A}}(e)$ as the expected probability, while the message frequency probability $F_M(e)$ is the observed probability. If M is of length L, then the observed frequency of $e \in \mathcal{A}$ is

$$O_M(e) = F_M(e) \cdot L$$

and the expected frequency of $e \in \mathcal{A}$ is

$$E_{\mathcal{A}}(e) = F_{\mathcal{A}}(e) \cdot L.$$

The squared-differences, or residual sum of squares, rank $R_{RSS}(M)$ of M corresponding to a key $(a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ is given by

$$R_{RSS}(M) = \sum_{e \in \mathcal{A}} \big(O_M(e) - E_{\mathcal{A}}(e)\big)^2.$$

Cryptanalysis by exhaustive key search produces a candidate decipherment $M_{a,b}$ for each possible key (a, b). For a set $D = \{M_{a_1,b_1}, M_{a_2,b_2}, \ldots, M_{a_k,b_k}\}$ of all candidate decipherments corresponding to a ciphertext C, the smaller is the rank $R_{RSS}(M_{a_i,b_i})$ the more likely that (a_i, b_i) is the secret key. This key ranking method is based on the residual sum of squares measure.

Input

- `C` — The ciphertext, a non-empty string. The ciphertext must be encoded using the upper-case letters of the English alphabet.
- `pdict` — A dictionary of key, possible plaintext pairs. This should be the output of `brute_force()` with `ranking="none"`.

Output

- A list ranking the most likely keys first. Each element of the list is a tuple of key, possible plaintext pairs.

Examples

Use the method of squared differences to rank all possible keys and their corresponding decipherment:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (3, 7)
sage: P = A.encoding("Line.")
sage: C = A.enciphering(a, b, P)
sage: Plist = A.brute_force(C)
sage: Rank = A.rank_by_squared_differences(C, Plist)
sage: Rank[:10]  # display only the top 10 candidate keys
<BLANKLINE>
[((1, 1), NETS),
((15, 6), ETUN),
((7, 17), HCTE),
((3, 7), LINE),
((17, 15), DELO),
((9, 4), EDWT),
((9, 9), POHE),
((21, 8), ELID),
((17, 20), STAD),
((7, 18), SNEP)]
```

As more ciphertext is available, the reliability of the squared-differences ranking function increases:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: a, b = (11, 24)
sage: P = A.encoding("Longer message is more information for cryptanalysis.")
sage: C = A.enciphering(a, b, P)
sage: Plist = A.brute_force(C)
sage: Rank = A.rank_by_squared_differences(C, Plist)
sage: Rank[:10]  # display only the top 10 candidate keys
<BLANKLINE>
[((11, 24), LONGERMESSAGEISMOREINFORMATIONFORCRYPTANALYSIS),
((9, 14), DYRUGTKGAAEUGIAKYTGIRNYTKEHIYRNYTSTQFHEREDQAIA),
((23, 24), DSNEUHIUMMAEUOMISHUONZSHIAROSNZSHKHQXRANADQMOM),
((23, 1), ETOFVIJVNNBFVPNJTIVPOATIJBSPTOATILIRYSBOBERNPN),
((21, 16), VEBGANYAQQOGAMQYENAMBDENYOTMEBDENUNIHTOBOVIQMQ),
((7, 12), TULAIVCIEEYAISECUVISLRUVCYNSULRUVQVGDNYLYTGESE),
((5, 20), ZQTOUHWUEEGOUIEWQHUITRQHWGBIQTRQHAHMNBGTGZMEIE),
((21, 8), JSPUOBMOEECUOAEMSBOAPRSBMCHASPRSBIBWVHCPCJWEAE),
((25, 7), SLWVREHRTTJVRZTHLERZWGLEHJIZLWGLENEFAIJWJSFTZT),
((25, 15), ATEDZMPZBBRDZHBPTMZHEOTMPRQHTEOTMVMNIQRERANBHB)]
```

Exception tests

The ciphertext cannot be an empty string:

```
sage: A.rank_by_squared_differences("", Plist)
...
AttributeError: 'str' object has no attribute 'parent'
sage: A.rank_by_squared_differences(A.encoding(""), Plist)
...
ValueError: The ciphertext must be a non-empty string.
sage: A.rank_by_squared_differences(A.encoding(" "), Plist)
...
ValueError: The ciphertext must be a non-empty string.
```

The ciphertext must be encoded using the capital letters of the English alphabet as implemented in `AlphabeticStrings()`:

```
sage: H = HexadecimalStrings()
sage: A.rank_by_squared_differences(H.encoding("line"), Plist)
...
TypeError: The ciphertext must be capital letters of the English
alphabet.
sage: B = BinaryStrings()
sage: A.rank_by_squared_differences(B.encoding("line"), Plist)
...
TypeError: The ciphertext must be capital letters of the English
alphabet.
```

The dictionary `pdict` cannot be empty:

```
sage: A.rank_by_squared_differences(C, {})
...
KeyError: (1, 0)
```

B.3 Private methods

This section documents private methods implemented in the class

`sage.crypto.classical.AffineCryptosystem`

of the Sage standard library.

B.3.1 `__init__(A)`

Construct an `AffineCryptosystem` object. See `AffineCryptosystem` for full documentation.

Input

- `A` — a string monoid over some alphabet; this is the non-empty alphabet over which the plaintext and ciphertext spaces are defined.

Output

- An affine cryptosystem over the alphabet `A`.

Examples

Testing of dumping and loading objects:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: A == loads(dumps(A))
True
```

B.3.2 `__call__(a, b)`

Create an affine cipher with secret key `(a, b)`.

Input

- `(a, b)` — a secret key; this key is used for both encryption and decryption. For the affine cryptosystem whose plaintext and ciphertext spaces are `A`, a key is an ordered pair $(a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ where n is the size or cardinality of the set `A` and $\gcd(a, n) = 1$.

Output

- An affine cipher with secret key `(a,b)`.

Examples

Creating an `AffineCryptosystem` object and perform cryptographic operations using it:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: P = A.encoding("Fine here, fine there."); P
FINEHEREFINETHERE
sage: a, b = (17, 3)
sage: E = A(a, b); E
Affine cipher on Free alphabetic string monoid on A-Z
sage: E(P)
KJQTSTGTKJQTOSTGT
sage: C = E(P)
sage: C
KJQTSTGTKJQTOSTGT
sage: aInv, bInv = A.inverse_key(a, b)
sage: D = A(aInv, bInv); D
Affine cipher on Free alphabetic string monoid on A-Z
sage: P == D(C)
True
sage: D(E(P))
FINEHEREFINETHERE
```

Exception tests

The key must be an ordered pair $(a, b) \in \mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$ with n being the size of the plaintext and ciphertext spaces. Furthermore, a must be relatively prime to n, i.e. $\gcd(a, n) = 1$:

```
sage: A = AffineCryptosystem(AlphabeticStrings())
sage: A(2, 3)
Traceback (most recent call last):
...
ValueError: (a, b) = (2, 3) is outside the range of acceptable values\
for a key of this affine cryptosystem.
```

B.3.3 _repr_()

Return a string representation of this affine cryptosystem.

Examples

```
sage: A = AffineCryptosystem(AlphabeticStrings()); A
Affine cryptosystem on Free alphabetic string monoid on A-Z
```

Appendix C

Sage Manual for Simplified DES

The S-DES symmetric-key cryptosystem described in Chapter 5 is implemented in the class

`sage.crypto.block_cipher.sdes.SimplifiedDES` (C.1)

via bug tracking ticket `#6461` [83]. This appendix provides the reference manual for the Sage class (C.1). The bug tracking ticket `#6461` has been merged in the Sage standard library during the development of Sage version 4.1.2.alpha0. The source code of the class (C.1) is available with the latest source release of Sage, which as of this writing is Sage version 4.2.1.

C.1 Class documentation

This class implements the Simplified Data Encryption Standard (S-DES) described in [102]. Schaefer's S-DES is for educational purposes only and is not secure for practical purposes. S-DES is a version of the DES with all parameters significantly reduced, but at the same time preserving the structure of DES. The goal of S-DES is to allow a beginner to understand the structure of DES, thus laying a foundation for a thorough study of DES. Its goal is as a teaching tool in the same spirit as Phan's Mini-AES [93].

C.1.1 Examples

Encrypt a random block of 8-bit plaintext using a random key, decrypt the ciphertext, and compare the result with the original plaintext:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES(); sdes
Simplified DES block cipher with 10-bit keys
sage: bin = BinaryStrings()
sage: P = [bin(str(randint(0, 1))) for i in xrange(8)]
sage: K = sdes.random_key()
sage: C = sdes.encrypt(P, K)
sage: plaintxt = sdes.decrypt(C, K)
sage: plaintxt == P
True
```

We can also encrypt binary strings that are larger than 8 bits in length. However, the number of bits in that binary string must be positive and a multiple of 8:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: bin = BinaryStrings()
sage: P = bin.encoding("Encrypt this using S-DES!")
sage: Mod(len(P), 8) == 0
True
sage: K = sdes.list_to_string(sdes.random_key())
sage: C = sdes(P, K, algorithm="encrypt")
sage: plaintxt = sdes(C, K, algorithm="decrypt")
sage: plaintxt == P
True
```

C.2 Public methods

This section documents public methods of the class

`sage.crypto.block_cipher.sdes.SimplifiedDES`

in the Sage standard library.

C.2.1 `block_length()`

Return the block length of Schaefer's S-DES block cipher. A key in Schaefer's S-DES is a block of 10 bits.

Output

- The block (or key) length in number of bits.

Examples

The block length of S-DES:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sdes.block_length()
10
```

C.2.2 `encrypt(P, K)`

Return an 8-bit ciphertext corresponding to the plaintext `P`, using S-DES encryption with key `K`. The encryption process of S-DES is as follows. Let P be the initial permutation function, P^{-1} the corresponding inverse permutation, Π_F the permutation/substitution function, and σ the switch function. The plaintext block P first goes through P, the output of which goes through Π_F using the first subkey. Then we apply the switch function to the output of the last function, and the result is then fed into Π_F using the second subkey. Finally, run the output through P^{-1} to get the ciphertext.

Input

- `P` — an 8-bit plaintext; a block of 8 bits.
- `K` — a 10-bit key; a block of 10 bits.

Output

- The 8-bit ciphertext corresponding to `P`, obtained using the key `K`.

Examples

Encrypt an 8-bit plaintext block:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: P = [0, 1, 0, 1, 0, 1, 0, 1]
sage: K = [1, 0, 1, 0, 0, 0, 0, 0, 1, 0]
sage: sdes.encrypt(P, K)
[1, 1, 0, 0, 0, 0, 0, 1]
```

We can also work with strings of bits:

```
sage: P = "01010101"
sage: K = "1010000010"
sage: sdes.encrypt(sdes.string_to_list(P), sdes.string_to_list(K))
[1, 1, 0, 0, 0, 0, 0, 1]
```

Exception tests

The plaintext must be a block of 8 bits:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sdes.encrypt("P", "K")
...
TypeError: plaintext must be a list of 8 bits
sage: sdes.encrypt([], "K")
...
ValueError: plaintext must be a list of 8 bits
sage: sdes.encrypt([1, 2, 3, 4], "K")
...
ValueError: plaintext must be a list of 8 bits
```

The key must be a block of 10 bits:

```
sage: sdes.encrypt([1, 0, 1, 0, 1, 1, 0, 1], "K")
...
TypeError: the key must be a list of 10 bits
sage: sdes.encrypt([1, 0, 1, 0, 1, 1, 0, 1], [])
...
TypeError: the key must be a list of 10 bits
sage: sdes.encrypt([1, 0, 1, 0, 1, 1, 0, 1], [1, 2, 3, 4, 5])
...
TypeError: the key must be a list of 10 bits
```

The value of each element of `P` or `K` must be either 0 or 1:

```
sage: P = [1, 2, 3, 4, 5, 6, 7, 8]
sage: K = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
sage: sdes.encrypt(P, K)
...
TypeError: Argument x (= 2) is not a valid string.
sage: P = [0, 1, 0, 0, 1, 1, 1, 0]
sage: K = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
sage: sdes.encrypt(P, K)
...
TypeError: Argument x (= 13) is not a valid string.
```

C.2.3 `decrypt(C, K)`

Return an 8-bit plaintext corresponding to the ciphertext `C`, using S-DES decryption with key `K`. The decryption process of S-DES is as follows. Let P be the initial permutation function, P^{-1} the corresponding inverse permutation, Π_F the permutation/substitution function, and σ the switch function. The ciphertext block C first goes through P, the output of which goes through Π_F using the second subkey. Then we apply the switch function to the output of the last function, and the result is then fed into Π_F using the first subkey. Finally, run the output through P^{-1} to get the plaintext.

Input

- `C` — an 8-bit ciphertext; a block of 8 bits.
- `K` — a 10-bit key; a block of 10 bits.

Output

- The 8-bit plaintext corresponding to `C`, obtained using the key `K`.

Examples

Decrypt an 8-bit ciphertext block:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: C = [0, 1, 0, 1, 0, 1, 0, 1]
sage: K = [1, 0, 1, 0, 0, 0, 0, 0, 1, 0]
sage: sdes.decrypt(C, K)
[0, 0, 0, 1, 0, 1, 0, 1]
```

We can also work with strings of bits:

```
sage: C = "01010101"
sage: K = "1010000010"
sage: sdes.decrypt(sdes.string_to_list(C), sdes.string_to_list(K))
[0, 0, 0, 1, 0, 1, 0, 1]
```

Exception tests

The ciphertext must be a block of 8 bits:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sdes.decrypt("C", "K")
...
TypeError: ciphertext must be a list of 8 bits
sage: sdes.decrypt([], "K")
...
ValueError: ciphertext must be a list of 8 bits
sage: sdes.decrypt([1, 2, 3, 4], "K")
...
ValueError: ciphertext must be a list of 8 bits
```

The key must be a block of 10 bits:

```
sage: sdes.decrypt([1, 0, 1, 0, 1, 1, 0, 1], "K")
...
TypeError: the key must be a list of 10 bits
sage: sdes.decrypt([1, 0, 1, 0, 1, 1, 0, 1], [])
...
TypeError: the key must be a list of 10 bits
sage: sdes.decrypt([1, 0, 1, 0, 1, 1, 0, 1], [1, 2, 3, 4, 5])
...
TypeError: the key must be a list of 10 bits
```

The value of each element of `C` or `K` must be either 0 or 1:

```
sage: C = [1, 2, 3, 4, 5, 6, 7, 8]
sage: K = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
sage: sdes.decrypt(C, K)
...
TypeError: Argument x (= 2) is not a valid string.
sage: C = [0, 1, 0, 0, 1, 1, 1, 0]
sage: K = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
sage: sdes.decrypt(C, K)
...
TypeError: Argument x (= 13) is not a valid string.
```

C.2.4 `initial_permutation(B, inverse=False)`

Return the initial permutation of `B`. Denote the initial permutation function by P and let $(b_0, b_1, b_2, \ldots, b_7)$ be a vector of 8 bits, where each $b_i \in \{0, 1\}$. Then

$$P(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7) = (b_1, b_5, b_2, b_0, b_3, b_7, b_4, b_6).$$

The inverse permutation is P^{-1}:

$$P^{-1}(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7) = (b_3, b_0, b_2, b_4, b_6, b_1, b_7, b_5).$$

Input

- `B` — list; a block of 8 bits.
- `inverse` — (default: `False`) if `True` then use the inverse permutation P^{-1}; if `False` then use the initial permutation `P`.

Output

- The initial permutation of `B` if `inverse=False`, or the inverse permutation of `B` if `inverse=True`.

Examples

The initial permutation of a list of 8 bits:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: B = [1, 0, 1, 1, 0, 1, 0, 0]
sage: P = sdes.initial_permutation(B); P
[0, 1, 1, 1, 1, 0, 0, 0]
```

Recovering the original list of 8 bits from the permutation:

```
sage: Pinv = sdes.initial_permutation(P, inverse=True)
sage: Pinv; B
[1, 0, 1, 1, 0, 1, 0, 0]
[1, 0, 1, 1, 0, 1, 0, 0]
```

We can also work with a string of bits:

```
sage: S = "10110100"
sage: L = sdes.string_to_list(S)
sage: P = sdes.initial_permutation(L); P
[0, 1, 1, 1, 1, 0, 0, 0]
sage: sdes.initial_permutation(sdes.string_to_list("01111000"), inverse=True)
[1, 0, 1, 1, 0, 1, 0, 0]
```

Exception tests

The input block must be a list:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sdes.initial_permutation("B")
...
TypeError: input block must be a list of 8 bits
sage: sdes.initial_permutation(())
...
TypeError: input block must be a list of 8 bits
```

The input block must be a list of 8 bits:

```
sage: sdes.initial_permutation([])
...
ValueError: input block must be a list of 8 bits
sage: sdes.initial_permutation([1, 2, 3, 4, 5, 6, 7, 8, 9])
...
ValueError: input block must be a list of 8 bits
```

The value of each element of the list must be either 0 or 1:

```
sage: sdes.initial_permutation([1, 2, 3, 4, 5, 6, 7, 8])
...
TypeError: Argument x (= 2) is not a valid string.
```

C.2.5 `left_shift(B, n=1)`

Return a circular left shift of `B` by `n` positions. Let $B = (b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9)$ be a vector of 10 bits. Then the left shift operation L_n is performed on the first 5 bits and the last 5 bits of B separately. That is, if the number of shift positions is $n = 1$, then L_1 is defined as

$$L_1(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9) = (b_1, b_2, b_3, b_4, b_0, b_6, b_7, b_8, b_9, b_5).$$

If the number of shift positions is $n = 2$, then L_2 is given by

$$L_2(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9) = (b_2, b_3, b_4, b_0, b_1, b_7, b_8, b_9, b_5, b_6).$$

Input

- `B` — a list of 10 bits.
- `n` — (default: `1`) if `n=1` then perform left shift by 1 position; if `n=2` then perform left shift by 2 positions. The valid values for `n` are `1` and `2`, since only up to 2 positions are defined for this circular left shift operation.

Output

- The circular left shift of each half of `B`.

Examples

Circular left shift by 1 position of a 10-bit string:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: B = [1, 0, 0, 0, 0, 0, 1, 1, 0, 0]
sage: sdes.left_shift(B)
[0, 0, 0, 0, 1, 1, 1, 0, 0, 0]
sage: sdes.left_shift([1, 0, 1, 0, 0, 0, 0, 0, 1, 0])
[0, 1, 0, 0, 1, 0, 0, 1, 0, 0]
```

Circular left shift by 2 positions of a 10-bit string:

```
sage: B = [0, 0, 0, 0, 1, 1, 1, 0, 0, 0]
sage: sdes.left_shift(B, n=2)
[0, 0, 1, 0, 0, 0, 0, 0, 1, 1]
```

Here we work with a string of bits:

```
sage: S = "1000001100"
sage: L = sdes.string_to_list(S)
sage: sdes.left_shift(L)
[0, 0, 0, 0, 1, 1, 1, 0, 0, 0]
sage: sdes.left_shift(sdes.string_to_list("1010000010"), n=2)
[1, 0, 0, 1, 0, 0, 1, 0, 0, 0]
```

Exception tests

The input block must be a list:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sdes.left_shift("B")
...
TypeError: input block must be a list of 10 bits
sage: sdes.left_shift(())
...
TypeError: input block must be a list of 10 bits
```

The input block must be a list of 10 bits:

```
sage: sdes.left_shift([])
...
ValueError: input block must be a list of 10 bits
sage: sdes.left_shift([1, 2, 3, 4, 5])
...
ValueError: input block must be a list of 10 bits
```

The value of each element of the list must be either 0 or 1:

```
sage: sdes.left_shift([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
...
TypeError: Argument x (= 2) is not a valid string.
```

The number of shift positions must be either 1 or 2:

```
sage: B = [0, 0, 0, 0, 1, 1, 1, 0, 0, 0]
sage: sdes.left_shift(B, n=-1)
...
ValueError: input n must be either 1 or 2
sage: sdes.left_shift(B, n=3)
...
ValueError: input n must be either 1 or 2
```

C.2.6 `list_to_string(B)`

Return a binary string representation of the list `B`.

Input

- `B` — a non-empty list of bits.

Output

- The binary string representation of `B`.

Examples

A binary string representation of a list of bits:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: L = [0, 0, 0, 0, 1, 1, 0, 1, 0, 0]
sage: sdes.list_to_string(L)
0000110100
```

Exception tests

Input `B` must be a non-empty list:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sdes.list_to_string("L")
...
TypeError: input B must be a non-empty list of bits
sage: sdes.list_to_string([])
...
ValueError: input B must be a non-empty list of bits
```

Input must be a non-empty list of bits:

```
sage: sdes.list_to_string([0, 1, 2])
...
IndexError: tuple index out of range
```

C.2.7 `permutation10(B)`

Return a permutation of a 10-bit string. This is the permutation function P_{10} and is specified as follows. Let $(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9)$ be a vector of 10 bits where each $b_i \in \{0, 1\}$. Then P_{10} is given by

$$P_{10}(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9) = (b_2, b_4, b_1, b_6, b_3, b_9, b_0, b_8, b_7, b_5).$$

Input

- `B` — a block of 10-bit string.

Output

- A permutation of `B` using P_{10}.

Examples

Permute a 10-bit string:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: B = [1, 1, 0, 0, 1, 0, 0, 1, 0, 1]
sage: sdes.permutation10(B)
[0, 1, 1, 0, 0, 1, 1, 0, 1, 0]
sage: sdes.permutation10([0, 1, 1, 0, 1, 0, 0, 1, 0, 1])
[1, 1, 1, 0, 0, 1, 0, 0, 1, 0]
sage: sdes.permutation10([1, 0, 1, 0, 0, 0, 0, 0, 1, 0])
[1, 0, 0, 0, 0, 0, 1, 1, 0, 0]
```

Here we work with a string of bits:

```
sage: S = "1100100101"
sage: L = sdes.string_to_list(S)
sage: sdes.permutation10(L)
[0, 1, 1, 0, 0, 1, 1, 0, 1, 0]
sage: sdes.permutation10(sdes.string_to_list("0110100101"))
[1, 1, 1, 0, 0, 1, 0, 0, 1, 0]
```

Exception tests

The input block must be a list:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sdes.permutation10("B")
...
TypeError: input block must be a list of 10 bits
sage: sdes.permutation10(())
...
TypeError: input block must be a list of 10 bits
```

The input block must be a list of 10 bits:

```
sage: sdes.permutation10([])
...
ValueError: input block must be a list of 10 bits
sage: sdes.permutation10([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
...
ValueError: input block must be a list of 10 bits
```

The value of each element of the list must be either 0 or 1:

```
sage: sdes.permutation10([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
...
TypeError: Argument x (= 3) is not a valid string.
```

C.2.8 `permutation4(B)`

Return a permutation of a 4-bit string. This is the permutation P_4 and is specified as follows. Let (b_0, b_1, b_2, b_3) be a vector of 4 bits where each $b_i \in \{0, 1\}$. Then P_4 is defined by

$$P_4(b_0, b_1, b_2, b_3) = (b_1, b_3, b_2, b_0).$$

Input

- `B` — a block of 4-bit string.

Output

- A permutation of `B` using P_4.

Examples

Permute a 4-bit string:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: B = [1, 1, 0, 0]
sage: sdes.permutation4(B)
[1, 0, 0, 1]
sage: sdes.permutation4([0, 1, 0, 1])
[1, 1, 0, 0]
```

We can also work with a string of bits:

```
sage: S = "1100"
sage: L = sdes.string_to_list(S)
sage: sdes.permutation4(L)
[1, 0, 0, 1]
sage: sdes.permutation4(sdes.string_to_list("0101"))
[1, 1, 0, 0]
```

Exception tests

The input block must be a list:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sdes.permutation4("B")
...
TypeError: input block must be a list of 4 bits
sage: sdes.permutation4(())
...
TypeError: input block must be a list of 4 bits
```

The input block must be a list of 4 bits:

```
sage: sdes.permutation4([])
...
ValueError: input block must be a list of 4 bits
sage: sdes.permutation4([1, 2, 3, 4, 5])
...
ValueError: input block must be a list of 4 bits
```

The value of each element of the list must be either 0 or 1:

```
sage: sdes.permutation4([1, 2, 3, 4])
...
TypeError: Argument x (= 2) is not a valid string.
```

C.2.9 `permutation8(B)`

Return a permutation of an 8-bit string. This is the permutation P_8 and is specified as follows. Let $(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9)$ be a vector of 10 bits where each $b_i \in \{0, 1\}$. Then P_8 picks out 8 of those 10 bits and permutes those 8 bits:

$$P_8(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9) = (b_5, b_2, b_6, b_3, b_7, b_4, b_9, b_8).$$

Input

- `B` — a block of 10-bit string.

Output

- Pick out 8 of the 10 bits of `B` and permute those 8 bits.

Examples

Permute a 10-bit string:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: B = [1, 1, 0, 0, 1, 0, 0, 1, 0, 1]
sage: sdes.permutation8(B)
[0, 0, 0, 0, 1, 1, 1, 0]
sage: sdes.permutation8([0, 1, 1, 0, 1, 0, 0, 1, 0, 1])
[0, 1, 0, 0, 1, 1, 1, 0]
sage: sdes.permutation8([0, 0, 0, 0, 1, 1, 1, 0, 0, 0])
[1, 0, 1, 0, 0, 1, 0, 0]
```

We can also work with a string of bits:

```
sage: S = "1100100101"
sage: L = sdes.string_to_list(S)
sage: sdes.permutation8(L)
[0, 0, 0, 0, 1, 1, 1, 0]
sage: sdes.permutation8(sdes.string_to_list("0110100101"))
[0, 1, 0, 0, 1, 1, 1, 0]
```

Exception tests

The input block must be a list:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sdes.permutation8("B")
...
TypeError: input block must be a list of 10 bits
sage: sdes.permutation8(())
...
TypeError: input block must be a list of 10 bits
```

The input block must be a list of 10 bits:

```
sage: sdes.permutation8([])
...
ValueError: input block must be a list of 10 bits
sage: sdes.permutation8([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
...
ValueError: input block must be a list of 10 bits
```

The value of each element of the list must be either 0 or 1:

```
sage: sdes.permutation8([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
...
TypeError: Argument x (= 6) is not a valid string.
```

C.2.10 `permute_substitute(B, key)`

Apply the Feistel function Π_F on the block `B` using subkey `key`. Let

$$(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7)$$

be a vector of 8 bits where each $b_i \in \{0, 1\}$, let L and R be the leftmost 4 bits and rightmost 4 bits of B respectively, and let F be a function mapping 4-bit strings to 4-bit strings. Then

$$\Pi_F(L, R) = (L \oplus F(R, S), R)$$

where S is a subkey and $\oplus$ denotes the bit-wise exclusive-OR function.

The function F can be described as follows. Its 4-bit input block (n_0, n_1, n_2, n_3) is first expanded into an 8-bit block to become $(n_3, n_0, n_1, n_2, n_1, n_2, n_3, n_0)$. This is usually represented as follows

$$\begin{array}{c|cc|c} n_3 & n_0 & n_1 & n_2 \\ n_1 & n_2 & n_3 & n_0 \end{array}.$$

Let $K = (k_0, k_1, k_2, k_3, k_4, k_5, k_6, k_7)$ be an 8-bit subkey. Then K is added to the above expanded input block using exclusive-OR to produce

$$\begin{array}{c|cc|c} n_3 + k_0 & n_0 + k_1 & n_1 + k_2 & n_2 + k_3 \\ n_1 + k_4 & n_2 + k_5 & n_3 + k_6 & n_0 + k_7 \end{array} = \begin{array}{c|cc|c} p_{0,0} & p_{0,1} & p_{0,2} & p_{0,3} \\ p_{1,0} & p_{1,1} & p_{1,2} & p_{1,3} \end{array}.$$

Now read the first row as the 4-bit string $p_{0,0}p_{0,3}p_{0,1}p_{0,2}$ and input this 4-bit string through S-box S_0 to get a 2-bit output.

$S_0 =$

Input	Output	Input	Output
0000	01	1000	00
0001	00	1001	10
0010	11	1010	01
0011	10	1011	11
0100	11	1100	11
0101	10	1101	01
0110	01	1110	11
0111	00	1111	10

Next read the second row as the 4-bit string $p_{1,0}p_{1,3}p_{1,1}p_{1,2}$ and input this 4-bit string through S-box S_1 to get another 2-bit output.

$S_1 =$

Input	Output	Input	Output
0000	00	1000	11
0001	01	1001	00
0010	10	1010	01
0011	11	1011	00
0100	10	1100	10
0101	00	1101	01
0110	01	1110	00
0111	11	1111	11

Denote the 4 bits produced by S_0 and S_1 as $b_0b_1b_2b_3$. This 4-bit string undergoes another permutation called P_4 as follows:

$$P_4(b_0, b_1, b_2, b_3) = (b_1, b_3, b_2, b_0).$$

The output of P_4 is the output of the function F.

Input

- `B` — a list of 8 bits.
- `key` — an 8-bit subkey.

Output

- The result of applying the function Π_F to `B` with subkey `key`.

Examples

Applying the function Π_F to an 8-bit block and an 8-bit subkey:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: B = [1, 0, 1, 1, 1, 1, 0, 1]
sage: K = [1, 1, 0, 1, 0, 1, 0, 1]
sage: sdes.permute_substitute(B, K)
[1, 0, 1, 0, 1, 1, 0, 1]
```

We can also work with strings of bits:

```
sage: B = "10111101"
sage: K = "11010101"
sage: B = sdes.string_to_list(B); K = sdes.string_to_list(K)
sage: sdes.permute_substitute(B, K)
[1, 0, 1, 0, 1, 1, 0, 1]
```

Exception tests

The input `B` must be a block of 8 bits:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sdes.permute_substitute("B", "K")
...
TypeError: input B must be an 8-bit string
sage: sdes.permute_substitute([], "K")
...
ValueError: input B must be an 8-bit string
```

The input key must be an 8-bit subkey:

```
sage: sdes.permute_substitute([0, 1, 0, 0, 1, 1, 1, 0], "K")
...
TypeError: input key must be an 8-bit subkey
sage: sdes.permute_substitute([0, 1, 0, 0, 1, 1, 1, 0], [])
...
ValueError: input key must be an 8-bit subkey
```

The value of each element of `B` or key must be either 0 or 1:

```
sage: B = [1, 2, 3, 4, 5, 6, 7, 8]
sage: K = [0, 1, 2, 3, 4, 5, 6, 7]
sage: sdes.permute_substitute(B, K)
...
TypeError: Argument x (= 2) is not a valid string.
sage: B = [0, 1, 0, 0, 1, 1, 1, 0]
sage: K = [1, 2, 3, 4, 5, 6, 7, 8]
sage: sdes.permute_substitute(B, K)
...
TypeError: Argument x (= 2) is not a valid string.
```

C.2.11 `random_key()`

Return a random 10-bit key.

Examples

The size of each key is the same as the block size:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: key = sdes.random_key()
sage: len(key) == sdes.block_length()
True
```

C.2.12 `sbox()`

Return the S-boxes of simplified DES.

Examples

The S-boxes of S-DES:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sbox = sdes.sbox()
sage: sbox[0]; sbox[1]
(1, 0, 3, 2, 3, 2, 1, 0, 0, 2, 1, 3, 3, 1, 3, 2)
(0, 1, 2, 3, 2, 0, 1, 3, 3, 0, 1, 0, 2, 1, 0, 3)
```

C.2.13 `string_to_list(S)`

Return a list representation of the binary string `S`.

Input

- `S` — a string of bits.

Output

- A list representation of the string `S`.

Examples

A list representation of a string of bits:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: S = "0101010110"
sage: sdes.string_to_list(S)
[0, 1, 0, 1, 0, 1, 0, 1, 1, 0]
```

Exception tests

Input must be a non-empty string:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sdes.string_to_list("")
...
ValueError: input S must be a non-empty string of bits
sage: sdes.string_to_list(1)
...
TypeError: input S must be a non-empty string of bits
```

Input must be a non-empty string of bits:

```
sage: sdes.string_to_list("0123")
...
TypeError: Argument x (= 2) is not a valid string.
```

C.2.14 `subkey(K, n=1)`

Return the n-th subkey based on the key `K`.

Input

- `K` — a 10-bit secret key of this simplified DES.
- `n` — (default: `1`) if `n=1` then return the first subkey based on `K`; if `n=2` then return the second subkey. The valid values for `n` are `1` and `2`, since only two subkeys are defined for each secret key in Schaefer's S-DES.

Output

- The n-th subkey based on the secret key `K`.

Examples

Obtain the first subkey from a secret key:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: key = [1, 0, 1, 0, 0, 0, 0, 0, 1, 0]
sage: sdes.subkey(key, n=1)
[1, 0, 1, 0, 0, 1, 0, 0]
```

Obtain the second subkey from a secret key:

```
sage: key = [1, 0, 1, 0, 0, 0, 0, 0, 1, 0]
sage: sdes.subkey(key, n=2)
[0, 1, 0, 0, 0, 0, 1, 1]
```

We can also work with strings of bits:

```
sage: K = "1010010010"
sage: L = sdes.string_to_list(K)
sage: sdes.subkey(L, n=1)
[1, 0, 1, 0, 0, 1, 0, 1]
sage: sdes.subkey(sdes.string_to_list("0010010011"), n=2)
[0, 1, 1, 0, 1, 0, 1, 0]
```

Exception tests

Input `K` must be a 10-bit key:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sdes.subkey("K")
...
TypeError: input K must be a 10-bit key
sage: sdes.subkey([])
...
ValueError: input K must be a 10-bit key
```

There are only two subkeys:

```
sage: key = [1, 0, 1, 0, 0, 0, 0, 0, 1, 0]
sage: sdes.subkey(key, n=0)
...
ValueError: input n must be either 1 or 2
sage: sdes.subkey(key, n=3)
...
ValueError: input n must be either 1 or 2
```

C.2.15 `switch(B)`

Interchange the first 4 bits with the last 4 bits in the list `B` of 8 bits. Let

$$(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7)$$

be a vector of 8 bits, where each $b_i \in \{0, 1\}$. Then the switch function σ is given by

$$\sigma(b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7) = (b_4, b_5, b_6, b_7, b_0, b_1, b_2, b_3).$$

Input

- `B` — list; a block of 8 bits.

Output

- A block of the same dimension, but in which the first 4 bits from `B` has been switched for the last 4 bits in `B`.

Examples

Interchange the first 4 bits with the last 4 bits:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: B = [1, 1, 1, 0, 1, 0, 0, 0]
sage: sdes.switch(B)
[1, 0, 0, 0, 1, 1, 1, 0]
sage: sdes.switch([1, 1, 1, 1, 0, 0, 0, 0])
[0, 0, 0, 0, 1, 1, 1, 1]
```

We can also work with a string of bits:

```
sage: S = "11101000"
sage: L = sdes.string_to_list(S)
sage: sdes.switch(L)
[1, 0, 0, 0, 1, 1, 1, 0]
sage: sdes.switch(sdes.string_to_list("11110000"))
[0, 0, 0, 0, 1, 1, 1, 1]
```

Exception tests

The input block must be a list:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sdes.switch("B")
...
TypeError: input block must be a list of 8 bits
sage: sdes.switch(())
...
TypeError: input block must be a list of 8 bits
```

The input block must be a list of 8 bits:

```
sage: sdes.switch([])
...
ValueError: input block must be a list of 8 bits
sage: sdes.switch([1, 2, 3, 4, 5, 6, 7, 8, 9])
...
ValueError: input block must be a list of 8 bits
```

The value of each element of the list must be either 0 or 1:

```
sage: sdes.switch([1, 2, 3, 4, 5, 6, 7, 8])
...
TypeError: Argument x (= 5) is not a valid string.
```

C.3 Private methods

This section documents private methods of the class

`sage.crypto.block_cipher.sdes.SimplifiedDES`

in the Sage standard library.

C.3.1 `__init__()`

Construct a simplified variant of the Data Encryption Standard (DES).

Examples

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES(); sdes
Simplified DES block cipher with 10-bit keys
sage: B = BinaryStrings()
sage: P = [B(str(randint(0, 1))) for i in xrange(8)]
sage: K = sdes.random_key()
sage: C = sdes.encrypt(P, K)
sage: plaintxt = sdes.decrypt(C, K)
sage: plaintxt == P
True
```

C.3.2 `__call__(B, K, algorithm="encrypt")`

Apply S-DES encryption or decryption on the binary string `B` using the key `K`. The flag `algorithm` controls what action is to be performed on `B`.

Input

- `B` — a binary string, where the number of bits is positive and a multiple of 8.
- `K` — a secret key; this must be a 10-bit binary string.
- `algorithm` — (default: `"encrypt"`) a string; a flag to signify whether encryption or decryption is to be applied to the binary string `B`. The encryption flag is `"encrypt"` and the decryption flag is `"decrypt"`.

Output

- The ciphertext (respectively plaintext) corresponding to the binary string `B`.

Examples

Encrypt a plaintext, decrypt the ciphertext, and compare the result with the original plaintext:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: bin = BinaryStrings()
sage: P = bin.encoding("Encrypt this using DES!")
sage: K = sdes.random_key()
sage: K = sdes.list_to_string(K)
sage: C = sdes(P, K, algorithm="encrypt")
sage: plaintxt = sdes(C, K, algorithm="decrypt")
sage: plaintxt == P
True
```

Exception tests

The binary string `B` must be non-empty and the number of bits must be a multiple of 8:

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: sdes = SimplifiedDES()
sage: sdes("B", "K")
Traceback (most recent call last):
...
TypeError: input B must be a non-empty binary string with number of\
bits a multiple of 8
sage: bin = BinaryStrings()
sage: B = bin("101")
sage: sdes(B, "K")
Traceback (most recent call last):
...
ValueError: the number of bits in the binary string B must be positive\
and a multiple of 8
```

The secret key `K` must be a block of 10 bits:

```
sage: B = bin.encoding("abc")
sage: sdes(B, "K")
Traceback (most recent call last):
...
TypeError: secret key must be a 10-bit binary string
sage: K = bin("1010")
sage: sdes(B, K)
Traceback (most recent call last):
...
ValueError: secret key must be a 10-bit binary string
```

The value for `algorithm` must be either `"encrypt"` or `"decrypt"`:

```
sage: B = bin.encoding("abc")
sage: K = sdes.list_to_string(sdes.random_key())
sage: sdes(B, K, algorithm="e")
Traceback (most recent call last):
...
ValueError: algorithm must be either 'encrypt' or 'decrypt'
sage: sdes(B, K, algorithm="d")
Traceback (most recent call last):
...
ValueError: algorithm must be either 'encrypt' or 'decrypt'
sage: sdes(B, K, algorithm="abc")
Traceback (most recent call last):
...
ValueError: algorithm must be either 'encrypt' or 'decrypt'
```

C.3.3 `__eq__(other)`

Compare whether or not this S-DES object is the same as the object in `other`.

Examples

Simplified DES objects are the same if they have the same key size and S-boxes.

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: s = SimplifiedDES()
sage: s == loads(dumps(s))
True
```

C.3.4 `__repr__()`

A string representation of this simplified DES.

Examples

```
sage: from sage.crypto.block_cipher.sdes import SimplifiedDES
sage: SimplifiedDES()
Simplified DES block cipher with 10-bit keys
```

Appendix D

Sage Manual for Mini-AES

The Mini-AES symmetric-key cryptosystem described in Chapter 6 is implemented in the class

`sage.crypto.block_cipher.miniaes.MiniAES` (D.1)

via bug tracking ticket `#6164` [80]. In this appendix, we provide the reference manual for the Sage class (D.1). The bug tracking ticket `#6164` has been merged in the Sage standard library during the development of Sage version 4.1.alpha2. The source code of the class (D.1) is available with the latest source release of Sage, which as of this writing is Sage version 4.2.1.

D.1 Class documentation

This class implements the Mini Advanced Encryption Standard (Mini-AES) described in [93]. Note that Phan's Mini-AES is for educational purposes only and is not secure for practical purposes. Mini-AES is a version of the AES with all parameters significantly reduced, but at the same time preserving the structure of AES. The goal of Mini-AES is to allow a beginner to understand the structure of AES, thus laying a foundation for a thorough study of AES. Its goal is as a teaching tool and is different from the SR small scale variants of the AES. SR defines a family of parameterizable variants of the AES suitable as a framework for comparing different cryptanalytic techniques that can be brought to bear on the AES.

D.1.1 Examples

Encrypt a plaintext:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: P = MS([K("x^3 + x"), K("x^2 + 1"), K("x^2 + x"), K("x^3 + x^2")]); P
<BLANKLINE>
[  x^3 + x   x^2 + 1]
[  x^2 + x x^3 + x^2]
sage: key = MS([K("x^3 + x^2"), K("x^3 + x"), K("x^3 + x^2 + x"), K("x^2 + x + 1")]); key
<BLANKLINE>
[    x^3 + x^2       x^3 + x]
[x^3 + x^2 + x   x^2 + x + 1]
sage: C = maes.encrypt(P, key); C
```

```
<BLANKLINE>
[            x       x^2 + x]
[x^3 + x^2 + x       x^3 + x]
```

Decrypt the result:

```
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt; P
<BLANKLINE>
[  x^3 + x   x^2 + 1]
[  x^2 + x x^3 + x^2]
<BLANKLINE>
[  x^3 + x   x^2 + 1]
[  x^2 + x x^3 + x^2]
sage: plaintxt == P
True
```

We can also work directly with binary strings:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: bin = BinaryStrings()
sage: key = bin.encoding("KE"); key
0100101101000101
sage: P = bin.encoding("Encrypt this secret message!"); P
010001010110111001100011011100100111100101110000011101000010000001110100011\
010000110100101110011001000000111001101100101011000110111001001100101011101\
000010000001101101011001010111001101110011011000010110011101110010100100001
sage: C = maes(P, key, algorithm="encrypt"); C
100010001010011011110000011110000100110011101101010001110110110101010010111\
011111010110011100111001000111011001010101000101001111101100110010100010001\
110110110100100000110001100011000001110000011100011010111100000000001110001001
sage: plaintxt = maes(C, key, algorithm="decrypt")
sage: plaintxt == P
True
```

Now we work with integers n such that $0 \leq n \leq 15$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: P = [n for n in xrange(16)]; P
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sage: key = [2, 3, 11, 0]; key
[2, 3, 11, 0]
sage: P = maes.integer_to_binary(P); P
0000000100100011010001010110011111000100110101011111001101111011111
sage: key = maes.integer_to_binary(key); key
0010001110110000
sage: C = maes(P, key, algorithm="encrypt"); C
110010000010001111100101010101010101101110011111100010000011100001
sage: plaintxt = maes(C, key, algorithm="decrypt")
sage: plaintxt == P
True
```

Generate some random plaintext and a random secret key. Encrypt the plaintext using that secret key and decrypt the result. Then compare the decrypted plaintext with the original plaintext:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: MS = MatrixSpace(FiniteField(16, "x"), 2, 2)
sage: P = MS.random_element()
sage: key = maes.random_key()
sage: C = maes.encrypt(P, key)
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt == P
True
```

D.2 Public methods

This section documents public methods in the class

`sage.crypto.block_cipher.miniaes.MiniAES`

of the Sage standard library.

D.2.1 `add_key(block, rkey)`

Return the matrix addition of `block` and `rkey`. Both `block` and `rkey` are 2×2 matrices over the finite field $\mathbf{F}_{2^4}[x]/(x^4+x^3+1)$. This method just return the matrix addition of these two matrices.

Input

- `block` — a 2×2 matrix with entries over $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$.
- `rkey` — a round key; a 2×2 matrix with entries over $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$.

Output

- The matrix addition of `block` and `rkey`.

Examples

We can work with elements of $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: D = MS([ [K("x^3 + x^2 + x + 1"), K("x^3 + x")], [K("0"), K("x^3 + x^2")] ]); D
<BLANKLINE>
[x^3 + x^2 + x + 1           x^3 + x]
[                0         x^3 + x^2]
sage: k = MS([ [K("x^2 + 1"), K("x^3 + x^2 + x + 1")], [K("x + 1"), K("0")] ]); k
<BLANKLINE>
[          x^2 + 1 x^3 + x^2 + x + 1]
[            x + 1                 0]
sage: maes.add_key(D, k)
<BLANKLINE>
[  x^3 + x   x^2 + 1]
[    x + 1 x^3 + x^2]
```

Or work with binary strings:

```
sage: bin = BinaryStrings()
sage: B = bin.encoding("We"); B
0101011101100101
sage: B = MS(maes.binary_to_GF(B)); B
<BLANKLINE>
[    x^2 + 1 x^2 + x + 1]
[    x^2 + x     x^2 + 1]
sage: key = bin.encoding("KY"); key
0100101101011001
sage: key = MS(maes.binary_to_GF(key)); key
<BLANKLINE>
[        x^2 x^3 + x + 1]
[    x^2 + 1     x^3 + 1]
sage: maes.add_key(B, key)
<BLANKLINE>
[        1 x^3 + x^2]
[    x + 1 x^3 + x^2]
```

We can also work with integers n such that $0 \leq n \leq 15$:

```
sage: N = [2, 3, 5, 7]; N
[2, 3, 5, 7]
sage: key = [9, 11, 13, 15]; key
[9, 11, 13, 15]
sage: N = MS(maes.integer_to_GF(N)); N
<BLANKLINE>
[          x       x + 1]
[    x^2 + 1 x^2 + x + 1]
sage: key = MS(maes.integer_to_GF(key)); key
<BLANKLINE>
[          x^3 + 1       x^3 + x + 1]
[    x^3 + x^2 + 1 x^3 + x^2 + x + 1]
sage: maes.add_key(N, key)
<BLANKLINE>
[x^3 + x + 1         x^3]
[        x^3         x^3]
```

Exception tests

The input block and key must each be a matrix:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MSB = MatrixSpace(K, 2, 2)
sage: B = MSB([ [K("x^3 + 1"), K("x^2 + x")], [K("x^3 + x^2"), K("x + 1")] ])
sage: maes.add_key(B, "key")
...
TypeError: round key must be a 2 x 2 matrix over GF(16)
sage: maes.add_key("block", "key")
...
TypeError: input block must be a 2 x 2 matrix over GF(16)
```

In addition, the dimensions of the input matrices must each be 2×2:

```
sage: MSB = MatrixSpace(K, 1, 2)
sage: B = MSB([ [K("x^3 + 1"), K("x^2 + x")] ])
sage: maes.add_key(B, "key")
...
TypeError: input block must be a 2 x 2 matrix over GF(16)
sage: MSB = MatrixSpace(K, 2, 2)
sage: B = MSB([ [K("x^3 + 1"), K("x^2 + x")], [K("x^3 + x^2"), K("x + 1")] ])
sage: MSK = MatrixSpace(K, 1, 2)
sage: key = MSK([ [K("x^3 + x^2"), K("x^3 + x^2 + x + 1")]])
sage: maes.add_key(B, key)
...
TypeError: round key must be a 2 x 2 matrix over GF(16)
```

D.2.2 `binary_to_GF(B)`

Return a list of elements of $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$ that represents the binary string `B`. The number of bits in `B` must be greater than zero and a multiple of 4. Each nibble (or 4-bit string) is uniquely associated with an element of $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$ as specified by the following table:

4-bit string	finite field element	4-bit string	finite field element
0000	0	1000	x^3
0001	1	1001	$x^3 + 1$
0010	x	1010	$x^3 + x$
0011	$x + 1$	1011	$x^3 + x + 1$
0100	x^2	1100	$x^3 + x^2$
0101	$x^2 + 1$	1101	$x^3 + x^2 + 1$
0110	$x^2 + x$	1110	$x^3 + x^2 + x$
0111	$x^2 + x + 1$	1111	$x^3 + x^2 + x + 1$

Input

- `B` — a binary string, where the number of bits is positive and a multiple of 4.

Output

- A list of elements of the finite field $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$ that represent the binary string `B`.

Examples

Obtain all the elements of the finite field $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: bin = BinaryStrings()
sage: B = bin("00000001001000110100010101100111100010011010101111001101111011111")
sage: maes.binary_to_GF(B)
<BLANKLINE>
[0,
1,
x,
x + 1,
x^2,
x^2 + 1,
x^2 + x,
```

```
x^2 + x + 1,
x^3,
x^3 + 1,
x^3 + x,
x^3 + x + 1,
x^3 + x^2,
x^3 + x^2 + 1,
x^3 + x^2 + x,
x^3 + x^2 + x + 1]
```

Exception tests

The input `B` must be a non-empty binary string, where the number of bits is a multiple of 4:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes.binary_to_GF("")
...
ValueError: the number of bits in the binary string B must be positive\
and a multiple of 4
sage: maes.binary_to_GF("101")
...
ValueError: the number of bits in the binary string B must be positive\
and a multiple of 4
```

D.2.3 `binary_to_integer(B)`

Return a list of integers representing the binary string `B`. The number of bits in `B` must be greater than zero and a multiple of 4. Each nibble (or 4-bit string) is uniquely associated with an integer as specified by the following table:

4-bit string	integer	4-bit string	integer
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

Input

- `B` — a binary string, where the number of bits is positive and a multiple of 4.

Output

- A list of integers that represent the binary string `B`.

Examples

Obtain the integer representation of every 4-bit string:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: bin = BinaryStrings()
sage: B = bin("0000000100100011010001010110011111000100110101011110011011110111101111")
sage: maes.binary_to_integer(B)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Exception tests

The input `B` must be a non-empty binary string, where the number of bits is a multiple of 4:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes.binary_to_integer("")
...
ValueError: the number of bits in the binary string B must be positive\
and a multiple of 4
sage: maes.binary_to_integer("101")
...
ValueError: the number of bits in the binary string B must be positive\
and a multiple of 4
```

D.2.4 `block_length()`

Return the block length of Phan's Mini-AES block cipher. A key in Phan's Mini-AES is a block of 16 bits. Each nibble of a key can be considered as an element of the finite field $\mathbf{F}_{2^4}[x]/(x^4+x^3+1)$. Therefore the key consists of four elements from $\mathbf{F}_{2^4}[x]/(x^4+x^3+1)$.

Output

- The block (or key) length in number of bits.

Examples

Obtaining the block length of Mini-AES:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes.block_length()
16
```

D.2.5 `encrypt(P, key)`

Use Phan's Mini-AES to encrypt the plaintext `P` with the secret key `key`. Both `P` and `key` must be 2×2 matrices over the finite field $\mathbf{F}_{2^4}[x]/(x^4+x^3+1)$. Let γ denote the operation of nibble-sub, π denote shift-row, θ denote mix-column, and σ_{K_i} denote add-key with the round key K_i. Then encryption `E` using Phan's Mini-AES is the function composition

$$E = \sigma_{K_2} \circ \pi \circ \gamma \circ \sigma_{K_1} \circ \theta \circ \pi \circ \gamma \circ \sigma_{K_0}$$

where the order of execution is from right to left. Note that γ is the nibble-sub operation that uses the S-box for encryption.

Input

- `P` — a plaintext block; must be a 2×2 matrix over the finite field $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$.
- `key` — a secret key for this Mini-AES block cipher; must be a 2×2 matrix over the finite field $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$.

Output

- The ciphertext corresponding to `P`.

Examples

Here we work with elements of $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: P = MS([ [K("x^3 + 1"), K("x^2 + x")], [K("x^3 + x^2"), K("x + 1")] ]); P
<BLANKLINE>
[  x^3 + 1   x^2 + x]
[x^3 + x^2     x + 1]
sage: key = MS([ [K("x^3 + x^2"), K("x^3 + x^2 + x + 1")], [K("x + 1"), K("0")] ]); key
<BLANKLINE>
[        x^3 + x^2 x^3 + x^2 + x + 1]
[            x + 1                 0]
sage: maes.encrypt(P, key)
<BLANKLINE>
[x^2 + x + 1   x^3 + x^2]
[          x     x^2 + x]
```

But we can also work with binary strings:

```
sage: bin = BinaryStrings()
sage: P = bin.encoding("de"); P
0110010001100101
sage: P = MS(maes.binary_to_GF(P)); P
<BLANKLINE>
[x^2 + x     x^2]
[x^2 + x x^2 + 1]
sage: key = bin.encoding("ke"); key
0110101101100101
```

```
sage: key = MS(maes.binary_to_GF(key)); key
<BLANKLINE>
[    x^2 + x x^3 + x + 1]
[    x^2 + x     x^2 + 1]
sage: C = maes.encrypt(P, key)
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt == P
True
```

Now we work with integers n such that $0 \leq n \leq 15$:

```
sage: P = [1, 5, 8, 12]; P
[1, 5, 8, 12]
sage: key = [5, 9, 15, 0]; key
[5, 9, 15, 0]
sage: P = MS(maes.integer_to_GF(P)); P
<BLANKLINE>
[        1   x^2 + 1]
[      x^3 x^3 + x^2]
sage: key = MS(maes.integer_to_GF(key)); key
<BLANKLINE>
[          x^2 + 1           x^3 + 1]
[x^3 + x^2 + x + 1                 0]
sage: C = maes.encrypt(P, key)
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt == P
True
```

Exception tests

The input block must be a matrix:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: key = MS([ [K("x^3 + x^2"), K("x^3 + x^2 + x + 1")], [K("x + 1"), K("0")] ])
sage: maes.encrypt("P", key)
...
TypeError: plaintext block must be a 2 x 2 matrix over GF(16)
sage: P = MS([ [K("x^3 + 1"), K("x^2 + x")], [K("x^3 + x^2"), K("x + 1")] ])
sage: maes.encrypt(P, "key")
...
TypeError: secret key must be a 2 x 2 matrix over GF(16)
```

In addition, the dimensions of the input matrices must be 2×2:

```
sage: MS = MatrixSpace(K, 1, 2)
sage: P = MS([ [K("x^3 + 1"), K("x^2 + x")]])
sage: maes.encrypt(P, "key")
...
TypeError: plaintext block must be a 2 x 2 matrix over GF(16)
sage: MSP = MatrixSpace(K, 2, 2)
sage: P = MSP([ [K("x^3 + 1"), K("x^2 + x")], [K("x^3 + x^2"), K("x + 1")] ])
sage: MSK = MatrixSpace(K, 1, 2)
sage: key = MSK([ [K("x^3 + x^2"), K("x^3 + x^2 + x + 1")]])
sage: maes.encrypt(P, key)
...
TypeError: secret key must be a 2 x 2 matrix over GF(16)
```

D.2.6 `decrypt(C, key)`

Use Phan's Mini-AES to decrypt the ciphertext `C` with the secret key `key`. Both `C` and `key` must be 2×2 matrices over the finite field $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$. Let γ denote the operation of nibble-sub, π denote shift-row, θ denote mix-column, and σ_{K_i} denote add-key with the round key K_i. Then decryption `D` using Phan's Mini-AES is the function composition

$$D = \sigma_{K_0} \circ \gamma^{-1} \circ \pi \circ \theta \circ \sigma_{K_1} \circ \gamma^{-1} \circ \pi \circ \sigma_{K_2}$$

where γ^{-1} is the nibble-sub operation that uses the S-box for decryption, and the order of execution is from right to left.

Input

- `C` — a ciphertext block; must be a 2×2 matrix over the finite field $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$.
- `key` — a secret key for this Mini-AES block cipher; must be a 2×2 matrix over the finite field $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$.

Output

- The plaintext corresponding to `C`.

Examples

We encrypt a plaintext, decrypt the ciphertext, then compare the decrypted plaintext with the original plaintext. Here we work with elements of $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: P = MS([ [K("x^3 + 1"), K("x^2 + x")], [K("x^3 + x^2"), K("x + 1")] ]); P
<BLANKLINE>
[  x^3 + 1   x^2 + x]
[x^3 + x^2     x + 1]
sage: key = MS([ [K("x^3 + x^2"), K("x^3 + x^2 + x + 1")], [K("x + 1"), K("0")] ]); key
<BLANKLINE>
[        x^3 + x^2 x^3 + x^2 + x + 1]
[            x + 1                 0]
sage: C = maes.encrypt(P, key); C
<BLANKLINE>
[x^2 + x + 1   x^3 + x^2]
[          x     x^2 + x]
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt; P
<BLANKLINE>
[  x^3 + 1   x^2 + x]
[x^3 + x^2     x + 1]
<BLANKLINE>
[  x^3 + 1   x^2 + x]
[x^3 + x^2     x + 1]
sage: plaintxt == P
True
```

But we can also work with binary strings:

```
sage: bin = BinaryStrings()
sage: P = bin.encoding("de"); P
0110010001100101
sage: P = MS(maes.binary_to_GF(P)); P
<BLANKLINE>
[x^2 + x     x^2]
[x^2 + x x^2 + 1]
sage: key = bin.encoding("ke"); key
0110101101100101
sage: key = MS(maes.binary_to_GF(key)); key
<BLANKLINE>
[    x^2 + x x^3 + x + 1]
[    x^2 + x     x^2 + 1]
sage: C = maes.encrypt(P, key)
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt == P
True
```

Here we work with integers n such that $0 \leq n \leq 15$:

```
sage: P = [3, 5, 7, 14]; P
[3, 5, 7, 14]
sage: key = [2, 6, 7, 8]; key
[2, 6, 7, 8]
sage: P = MS(maes.integer_to_GF(P)); P
<BLANKLINE>
[        x + 1       x^2 + 1]
[  x^2 + x + 1 x^3 + x^2 + x]
sage: key = MS(maes.integer_to_GF(key)); key
<BLANKLINE>
[          x     x^2 + x]
[x^2 + x + 1         x^3]
sage: C = maes.encrypt(P, key)
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt == P
True
```

Exception tests

The input block must be a matrix:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: key = MS([ [K("x^3 + x^2"), K("x^3 + x^2 + x + 1")], [K("x + 1"), K("0")] ])
sage: maes.decrypt("C", key)
...
TypeError: ciphertext block must be a 2 x 2 matrix over GF(16)
sage: C = MS([ [K("x^3 + 1"), K("x^2 + x")], [K("x^3 + x^2"), K("x + 1")] ])
sage: maes.decrypt(C, "key")
...
TypeError: secret key must be a 2 x 2 matrix over GF(16)
```

In addition, the dimensions of the input matrices must be 2×2:

```
sage: MS = MatrixSpace(K, 1, 2)
sage: C = MS([ [K("x^3 + 1"), K("x^2 + x")]])
sage: maes.decrypt(C, "key")
...
```

```
TypeError: ciphertext block must be a 2 x 2 matrix over GF(16)
sage: MSC = MatrixSpace(K, 2, 2)
sage: C = MSC([ [K("x^3 + 1"), K("x^2 + x")], [K("x^3 + x^2"), K("x + 1")] ])
sage: MSK = MatrixSpace(K, 1, 2)
sage: key = MSK([ [K("x^3 + x^2"), K("x^3 + x^2 + x + 1")]])
sage: maes.decrypt(C, key)
...
TypeError: secret key must be a 2 x 2 matrix over GF(16)
```

D.2.7 `GF_to_binary(G)`

Return the binary representation of `G`. If `G` is an element of the finite field $\mathbf{F}_{2^4}[x]/(x^4+x^3+1)$, then obtain the binary representation of `G`. If `G` is a list of elements belonging to $\mathbf{F}_{2^4}[x]/(x^4+x^3+1)$, obtain the 4-bit representation of each element of the list, then concatenate the resulting 4-bit strings into a binary string. If `G` is a matrix with entries over $\mathbf{F}_{2^4}[x]/(x^4+x^3+1)$, convert each matrix entry to its 4-bit representation, then concatenate the 4-bit strings. The concatenation is performed starting from the top-left corner of the matrix, working across left to right, top to bottom. Each element of $\mathbf{F}_{2^4}[x]/(x^4+x^3+1)$ can be associated with a unique 4-bit string according to the following table:

4-bit string	finite field element	4-bit string	finite field element
0000	0	1000	x^3
0001	1	1001	x^3+1
0010	x	1010	x^3+x
0011	$x+1$	1011	x^3+x+1
0100	x^2	1100	x^3+x^2
0101	x^2+1	1101	x^3+x^2+1
0110	x^2+x	1110	x^3+x^2+x
0111	x^2+x+1	1111	x^3+x^2+x+1

Input

- `G` — an element of $\mathbf{F}_{2^4}[x]/(x^4+x^3+1)$, a list of elements of $\mathbf{F}_{2^4}[x]/(x^4+x^3+1)$, or a matrix over $\mathbf{F}_{2^4}[x]/(x^4+x^3+1)$.

Output

- A binary string representation of `G`.

Examples

Obtain the binary representation of all elements of $\mathbf{F}_{2^4}[x]/(x^4+x^3+1)$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: S = Set(K); len(S)  # GF(2^4) has this many elements
16
sage: [maes.GF_to_binary(S[i]) for i in xrange(len(S))]
<BLANKLINE>
[0000,
0001,
0010,
```

```
0011,
0100,
0101,
0110,
0111,
1000,
1001,
1010,
1011,
1100,
1101,
1110,
1111]
```

The binary representation of a list of elements belonging to $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: G = [K("x^2 + x + 1"), K("x^3 + x^2"), K("x"),\
  K("x^3 + x + 1"), K("x^3 + x^2 + x + 1"), K("x^2 + x"),\
  K("1"), K("x^2 + x + 1")]
sage: maes.GF_to_binary(G)
011111000010101111111011000010111
```

The binary representation of a matrix over $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: G = MS([K("x^3 + x^2"), K("x + 1"), K("x^2 + x + 1"), K("x^3 + x^2 + x")]); G
<BLANKLINE>
[    x^3 + x^2         x + 1]
[  x^2 + x + 1 x^3 + x^2 + x]
sage: maes.GF_to_binary(G)
1100001101111110
sage: MS = MatrixSpace(K, 2, 4)
sage: G = MS([K("x^2 + x + 1"), K("x^3 + x^2"), K("x"),\
  K("x^3 + x + 1"), K("x^3 + x^2 + x + 1"), K("x^2 + x"),\
  K("1"), K("x^2 + x + 1")]); G
<BLANKLINE>
[      x^2 + x + 1         x^3 + x^2                 x       x^3 + x + 1]
[x^3 + x^2 + x + 1           x^2 + x                 1       x^2 + x + 1]
sage: maes.GF_to_binary(G)
011111000010101111111011000010111
```

Exception tests

Input must be an element of $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(8, "x")
sage: G = K.random_element()
sage: maes.GF_to_binary(G)
...
TypeError: input G must be an element of GF(16), a list of elements of\
GF(16), or a matrix over GF(16)
```

A list of elements belonging to $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: maes.GF_to_binary([])
...
ValueError: input G must be an element of GF(16), a list of elements\
of GF(16), or a matrix over GF(16)
sage: G = [K.random_element() for i in xrange(5)]
sage: maes.GF_to_binary(G)
...
KeyError:...
```

A matrix over $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: MS = MatrixSpace(FiniteField(7, "x"), 4, 5)
sage: maes.GF_to_binary(MS.random_element())
...
TypeError: input G must be an element of GF(16), a list of elements of\
GF(16), or a matrix over GF(16)
```

D.2.8 `GF_to_integer(G)`

Return the integer representation of the finite field element `G`. If `G` is an element of the finite field $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$, then obtain the integer representation of `G`. If `G` is a list of elements belonging to $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$, obtain the integer representation of each element of the list, and return the result as a list of integers. If G is a matrix with entries over $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$, convert each matrix entry to its integer representation, and return the result as a list of integers. The resulting list is obtained by starting from the top-left corner of the matrix, working across left to right, top to bottom. Each element of $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$ can be associated with a unique integer according to the following table:

integer	finite field element	integer	finite field element
0	0	8	x^3
1	1	9	$x^3 + 1$
2	x	10	$x^3 + x$
3	$x + 1$	11	$x^3 + x + 1$
4	x^2	12	$x^3 + x^2$
5	$x^2 + 1$	13	$x^3 + x^2 + 1$
6	$x^2 + x$	14	$x^3 + x^2 + x$
7	$x^2 + x + 1$	15	$x^3 + x^2 + x + 1$

Input

- `G` — an element of $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$, a list of elements belonging to $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$, or a matrix over $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$.

Output

- The integer representation of `G`.

Examples

Obtain the integer representation of all elements of $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: S = Set(K); len(S)  # GF(2^4) has this many elements
16
sage: [maes.GF_to_integer(S[i]) for i in xrange(len(S))]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

The integer representation of a list of elements belonging to $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: G = [K("x^2 + x + 1"), K("x^3 + x^2"), K("x"),\
  K("x^3 + x + 1"), K("x^3 + x^2 + x + 1"), K("x^2 + x"),\
  K("1"), K("x^2 + x + 1")]
sage: maes.GF_to_integer(G)
[7, 12, 2, 11, 15, 6, 1, 7]
```

The integer representation of a matrix over $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: G = MS([K("x^3 + x^2"), K("x + 1"), K("x^2 + x + 1"), K("x^3 + x^2 + x")]); G
<BLANKLINE>
[    x^3 + x^2         x + 1]
[  x^2 + x + 1 x^3 + x^2 + x]
sage: maes.GF_to_integer(G)
[12, 3, 7, 14]
sage: MS = MatrixSpace(K, 2, 4)
sage: G = MS([K("x^2 + x + 1"), K("x^3 + x^2"), K("x"),\
  K("x^3 + x + 1"), K("x^3 + x^2 + x + 1"), K("x^2 + x"),\
  K("1"), K("x^2 + x + 1")]); G
<BLANKLINE>
[      x^2 + x + 1         x^3 + x^2                 x       x^3 + x + 1]
[x^3 + x^2 + x + 1           x^2 + x                 1       x^2 + x + 1]
sage: maes.GF_to_integer(G)
[7, 12, 2, 11, 15, 6, 1, 7]
```

Exception tests

Input must be an element of $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(7, "x")
sage: G = K.random_element()
sage: maes.GF_to_integer(G)
...
TypeError: input G must be an element of GF(16), a list of elements of\
GF(16), or a matrix over GF(16)
```

A list of elements belonging to $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: maes.GF_to_integer([])
...
ValueError: input G must be an element of GF(16), a list of elements\
of GF(16), or a matrix over GF(16)
sage: G = [K.random_element() for i in xrange(5)]
sage: maes.GF_to_integer(G)
...
KeyError:...
```

A matrix over $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: MS = MatrixSpace(FiniteField(7, "x"), 4, 5)
sage: maes.GF_to_integer(MS.random_element())
...
TypeError: input G must be an element of GF(16), a list of elements of\
GF(16), or a matrix over GF(16)
```

D.2.9 integer_to_binary(N)

Return the binary representation of `N`. If `N` is an integer such that $0 \leq N \leq 15$, return the binary representation of `N`. If `N` is a list of integers each of which is ≥ 0 and ≤ 15, then obtain the binary representation of each integer, and concatenate the individual binary representations into a single binary string. Each integer between 0 and 15, inclusive, can be associated with a unique 4-bit string according to the following table:

4-bit string	integer	4-bit string	integer
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

Input

- `N` — a non-negative integer less than or equal to 15, or a list of such integers.

Output

- A binary string representing `N`.

Examples

The binary representations of all integers between 0 and 15, inclusive:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: lst = [n for n in xrange(16)]; lst
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sage: maes.integer_to_binary(lst)
0000000100100011010001010110011110001001101010111100110111101111
```

The binary representation of an integer between 0 and 15, inclusive:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes.integer_to_binary(3)
0011
sage: maes.integer_to_binary(5)
0101
sage: maes.integer_to_binary(7)
0111
```

Exception tests

The input `N` can be an integer, but must be bounded such that $0 \leq N \leq 15$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes.integer_to_binary(-1)
...
KeyError:...
sage: maes.integer_to_binary("1")
...
TypeError: N must be an integer 0 <= N <= 15 or a list of such integers
sage: maes.integer_to_binary(")
...
TypeError: N must be an integer 0 <= N <= 15 or a list of such integers
```

The input `N` can be a list of integers, but each integer n of the list must be $0 \leq n \leq 15$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes.integer_to_binary([])
...
ValueError: N must be an integer 0 <= N <= 15 or a list of such
integers
sage: maes.integer_to_binary([""])
...
KeyError:...
sage: maes.integer_to_binary([0, 1, 2, 16])
...
KeyError:...
```

D.2.10 `integer_to_GF(N)`

Return the finite field representation of `N`. If `N` is an integer such that $0 \leq N \leq 15$, return the element of $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$ that represents `N`. If `N` is a list of integers each of which is ≥ 0 and ≤ 15, then obtain the element of $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$ that represents each such integer, and return a list of such finite field representations. Each integer between 0 and 15, inclusive, can be associated with a unique element of $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$ according to the following table:

integer	finite field element	integer	finite field element
0	0	8	x^3
1	1	9	$x^3 + 1$
2	x	10	$x^3 + x$
3	$x + 1$	11	$x^3 + x + 1$
4	x^2	12	$x^3 + x^2$
5	$x^2 + 1$	13	$x^3 + x^2 + 1$
6	$x^2 + x$	14	$x^3 + x^2 + x$
7	$x^2 + x + 1$	15	$x^3 + x^2 + x + 1$

Input

- `N` — a non-negative integer less than or equal to 15, or a list of such integers.

Output

- Elements of the finite field $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$.

Examples

Obtain the element of $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$ representing an integer n, where $0 \leq n \leq 15$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes.integer_to_GF(0)
0
sage: maes.integer_to_GF(2)
x
sage: maes.integer_to_GF(7)
x^2 + x + 1
```

Obtain the finite field elements corresponding to all non-negative integers less than or equal to 15:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: lst = [n for n in xrange(16)]; lst
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sage: maes.integer_to_GF(lst)
<BLANKLINE>
[0,
1,
x,
x + 1,
x^2,
x^2 + 1,
x^2 + x,
x^2 + x + 1,
x^3,
x^3 + 1,
x^3 + x,
x^3 + x + 1,
x^3 + x^2,
x^3 + x^2 + 1,
x^3 + x^2 + x,
x^3 + x^2 + x + 1]
```

Exception tests

The input `N` can be an integer, but it must be such that $0 \leq N \leq 15$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes.integer_to_GF(-1)
...
KeyError:...
sage: maes.integer_to_GF(16)
...
KeyError:...
sage: maes.integer_to_GF("2")
...
TypeError: N must be an integer 0 <= N <= 15 or a list of such integers
```

The input `N` can be a list of integers, but each integer n in the list must be bounded such that $0 \leq n \leq 15$:

```
sage: maes.integer_to_GF([])
...
ValueError: N must be an integer 0 <= N <= 15 or a list of such integers
sage: maes.integer_to_GF(["])
...
KeyError:...
sage: maes.integer_to_GF([0, 2, 3, "4"])
...
KeyError:...
sage: maes.integer_to_GF([0, 2, 3, 16])
...
KeyError:...
```

D.2.11 `mix_column(block)`

Return the matrix multiplication of `block` with the matrix

$$\begin{bmatrix} x+1 & x \\ x & x+1 \end{bmatrix}.$$

If the input block is

$$\begin{bmatrix} c_0 & c_2 \\ c_1 & c_3 \end{bmatrix}$$

then the output block is

$$\begin{bmatrix} d_0 & d_2 \\ d_1 & d_3 \end{bmatrix} = \begin{bmatrix} x+1 & x \\ x & x+1 \end{bmatrix} \begin{bmatrix} c_0 & c_2 \\ c_1 & c_3 \end{bmatrix}.$$

Input

- `block` — a 2×2 matrix with entries over $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$.

Output

- A 2×2 matrix resulting from multiplying the above matrix with the input matrix block.

Examples

Here we work with elements of $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: mat = MS([ [K("x^2 + x + 1"), K("x^3 + x^2 + 1")], [K("x^3"), K("x")] ])
sage: maes.mix_column(mat)
<BLANKLINE>
[          x^3 + x                  0]
[          x^2 + 1 x^3 + x^2 + x + 1]
```

Multiplying by the identity matrix should leave the fixed matrix unchanged:

```
sage: eye = MS([ [K("1"), K("0")], [K("0"), K("1")] ])
sage: maes.mix_column(eye)
<BLANKLINE>
[x + 1     x]
[    x x + 1]
```

We can also work with binary strings:

```
sage: bin = BinaryStrings()
sage: B = bin.encoding("rT"); B
0111001001010100
sage: B = MS(maes.binary_to_GF(B)); B
<BLANKLINE>
[x^2 + x + 1           x]
[    x^2 + 1         x^2]
sage: maes.mix_column(B)
<BLANKLINE>
[        x + 1 x^3 + x^2 + x]
[            1           x^3]
```

We can also work with integers n such that $0 \leq n \leq 15$:

```
sage: P = [10, 5, 2, 7]; P
[10, 5, 2, 7]
sage: P = MS(maes.integer_to_GF(P)); P
<BLANKLINE>
[    x^3 + x     x^2 + 1]
[          x x^2 + x + 1]
sage: maes.mix_column(P)
<BLANKLINE>
[x^3 + 1       1]
[      1   x + 1]
```

Exception tests

The input block must be a matrix:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes.mix_column("mat")
...
TypeError: input block must be a 2 x 2 matrix over GF(16)
```

In addition, the dimensions of the input matrix must be 2×2:

```
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 1, 2)
sage: mat = MS([[K("x^3 + x^2 + x + 1"), K("0")]])
sage: maes.mix_column(mat)
...
TypeError: input block must be a 2 x 2 matrix over GF(16)
```

D.2.12 nibble_sub(block, algorithm='encrypt')

Substitute a nibble (or a block of 4 bits) using the S-box for encryption or decryption.

Input

- `block` — a 2×2 matrix with entries over $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$.
- `algorithm` — (default: `"encrypt"`) a string; a flag to signify whether this nibble-sub operation is used for encryption or decryption. The encryption flag is `"encrypt"` and the decryption flag is `"decrypt"`.

Output

- A 2×2 matrix resulting from applying an S-box on entries of the 2×2 matrix block.

Examples

Here we work with elements of the finite field $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: mat = MS([[K("x^3 + x^2 + x + 1"), K("0")], [K("x^2 + x + 1"), K("x^3 + x")]])
sage: maes.nibble_sub(mat, algorithm=''encrypt'')
<BLANKLINE>
[  x^2 + x + 1 x^3 + x^2 + x]
[          x^3       x^2 + x]
```

But we can also work with binary strings:

```
sage: bin = BinaryStrings()
sage: B = bin.encoding("bi"); B
0110001001101001
sage: B = MS(maes.binary_to_GF(B)); B
<BLANKLINE>
[x^2 + x       x]
[x^2 + x x^3 + 1]
sage: maes.nibble_sub(B, algorithm="encrypt")
<BLANKLINE>
[  x^3 + x + 1 x^3 + x^2 + 1]
[  x^3 + x + 1       x^3 + x]
sage: maes.nibble_sub(B, algorithm="decrypt")
<BLANKLINE>
[      x^3 + x           x^2]
[      x^3 + x x^3 + x^2 + 1]
```

Here we work with integers n such that $0 \leq n \leq 15$:

```
sage: P = [2, 6, 8, 14]; P
[2, 6, 8, 14]
sage: P = MS(maes.integer_to_GF(P)); P
<BLANKLINE>
[            x       x^2 + x]
[          x^3 x^3 + x^2 + x]
sage: maes.nibble_sub(P, algorithm="encrypt")
<BLANKLINE>
[x^3 + x^2 + 1   x^3 + x + 1]
[        x + 1             0]
sage: maes.nibble_sub(P, algorithm="decrypt")
<BLANKLINE>
[        x^2     x^3 + x]
[x^2 + x + 1           0]
```

Exception tests

The input block must be a matrix:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes.nibble_sub("mat")
...
TypeError: input block must be a 2 x 2 matrix over GF(16)
```

In addition, the dimensions of the input matrix must be 2×2:

```
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 1, 2)
sage: mat = MS([[K("x^3 + x^2 + x + 1"), K("0")]])
sage: maes.nibble_sub(mat)
...
TypeError: input block must be a 2 x 2 matrix over GF(16)
```

The value for the option `algorithm` must be either the string `"encrypt"` or `"decrypt"`:

```
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: mat = MS([[K("x^3 + x^2 + x + 1"), K("0")], [K("x^2 + x + 1"), K("x^3 + x")]])
sage: maes.nibble_sub(mat, algorithm="abc")
...
ValueError: the algorithm for nibble-sub must be either 'encrypt' or 'decrypt'
sage: maes.nibble_sub(mat, algorithm=''e'')
...
ValueError: the algorithm for nibble-sub must be either 'encrypt' or 'decrypt'
sage: maes.nibble_sub(mat, algorithm=''d'')
...
ValueError: the algorithm for nibble-sub must be either 'encrypt' or 'decrypt'
```

D.2.13 `random_key()`

A random key within the key space of this Mini-AES block cipher. Like the AES, Phan's Mini-AES is a symmetric-key block cipher. A Mini-AES key is a block of 16 bits, or a 2×2 matrix with entries over the finite field $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$. Thus the number of possible keys is $2^{16} = 16^4$.

Output

- A 2×2 matrix over the finite field $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$, used as a secret key for this Mini-AES block cipher.

Examples

Each nibble of a key is an element of the finite field $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: K = FiniteField(16, "x")
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: key = maes.random_key()
sage: [key[i][j] in K for i in xrange(key.nrows()) for j in xrange(key.ncols())]
[True, True, True, True]
```

Generate a random key, then perform encryption and decryption using that key:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: key = maes.random_key()
sage: P = MS.random_element()
sage: C = maes.encrypt(P, key)
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt == P
True
```

D.2.14 `round_key(key, n)`

Return the round key for round `n`. Phan's Mini-AES is defined to have two rounds. The round key K_0 is generated and used prior to the first round, with round keys K_1 and K_2 being used in rounds 1 and 2 respectively. In total, there are three round keys, each generated from the secret key `key`.

Input

- `key` — the secret key.
- `n` — non-negative integer; the round number.

Output

- The n-th round key.

Examples

Obtaining the round keys from the secret key:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: key = MS([ [K("x^3 + x^2"), K("x^3 + x^2 + x + 1")], [K("x + 1"), K("0")] ])
sage: maes.round_key(key, 0)
<BLANKLINE>
[        x^3 + x^2 x^3 + x^2 + x + 1]
[            x + 1                 0]
sage: key
<BLANKLINE>
[        x^3 + x^2 x^3 + x^2 + x + 1]
[            x + 1                 0]
sage: maes.round_key(key, 1)
<BLANKLINE>
[            x + 1 x^3 + x^2 + x + 1]
[                0 x^3 + x^2 + x + 1]
sage: maes.round_key(key, 2)
<BLANKLINE>
[x^2 + x x^3 + 1]
[x^2 + x x^2 + x]
```

Exception tests

Only two rounds are defined for this AES variant:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: key = MS([ [K("x^3 + x^2"), K("x^3 + x^2 + x + 1")], [K("x + 1"), K("0")] ])
sage: maes.round_key(key, -1)
...
ValueError: Mini-AES only defines two rounds
sage: maes.round_key(key, 3)
...
ValueError: Mini-AES only defines two rounds
```

The input key must be a matrix:

```
sage: maes.round_key("key", 0)
...
TypeError: secret key must be a 2 x 2 matrix over GF(16)
```

In addition, the dimensions of the key matrix must be 2×2:

```
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 1, 2)
sage: key = MS([[K("x^3 + x^2 + x + 1"), K("0")]])
sage: maes.round_key(key, 2)
...
TypeError: secret key must be a 2 x 2 matrix over GF(16)
```

D.2.15 `sbox()`

Return the S-box of Mini-AES.

Examples

Obtain the S-box of Mini-AES:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes.sbox()
(14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7)
```

D.2.16 `shift_row(block)`

Rotate each row of `block` to the left by different nibble amounts. The first or zero-th row is left unchanged, while the second or row one is rotated left by one nibble. This has the effect of only interchanging the nibbles in the second row. Let b_0, b_1, b_2, b_3 be four nibbles arranged as the following 2×2 matrix

$$\begin{bmatrix} b_0 & b_2 \\ b_1 & b_3 \end{bmatrix}.$$

Then the operation of shift-row is the mapping

$$\begin{bmatrix} b_0 & b_2 \\ b_1 & b_3 \end{bmatrix} \longmapsto \begin{bmatrix} b_0 & b_2 \\ b_3 & b_1 \end{bmatrix}.$$

Input

- `block` — a 2×2 matrix with entries over $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$.

Output

- A 2×2 matrix resulting from applying shift-row on `block`.

Examples

Here we work with elements of the finite field $\mathbf{F}_{2^4}[x]/(x^4 + x^3 + 1)$:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 2, 2)
sage: mat = MS([[K("x^3 + x^2 + x + 1"), K("0")], [K("x^2 + x + 1"), K("x^3 + x")]])
sage: maes.shift_row(mat)
<BLANKLINE>
[x^3 + x^2 + x + 1                0]
[         x^3 + x       x^2 + x + 1]
sage: mat
<BLANKLINE>
[x^3 + x^2 + x + 1                0]
[      x^2 + x + 1          x^3 + x]
```

But we can also work with binary strings:

```
sage: bin = BinaryStrings()
sage: B = bin.encoding("Qt"); B
0101000101110100
sage: B = MS(maes.binary_to_GF(B)); B
<BLANKLINE>
[    x^2 + 1           1]
[x^2 + x + 1         x^2]
sage: maes.shift_row(B)
<BLANKLINE>
[    x^2 + 1           1]
[        x^2 x^2 + x + 1]
```

Here we work with integers n such that $0 \leq n \leq 15$:

```
sage: P = [3, 6, 9, 12]; P
[3, 6, 9, 12]
sage: P = MS(maes.integer_to_GF(P)); P
<BLANKLINE>
[    x + 1   x^2 + x]
[  x^3 + 1 x^3 + x^2]
sage: maes.shift_row(P)
<BLANKLINE>
[    x + 1   x^2 + x]
[x^3 + x^2   x^3 + 1]
```

Exception tests

The input block must be a matrix:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes.shift_row("block")
...
TypeError: input block must be a 2 x 2 matrix over GF(16)
```

In addition, the dimensions of the input matrix must be 2×2:

```
sage: K = FiniteField(16, "x")
sage: MS = MatrixSpace(K, 1, 2)
sage: mat = MS([[K("x^3 + x^2 + x + 1"), K("0")]])
sage: maes.shift_row(mat)
...
TypeError: input block must be a 2 x 2 matrix over GF(16)
```

D.3 Private methods

This section documents private methods of the class

`sage.crypto.block_cipher.miniaes.MiniAES`

in the Sage standard library.

D.3.1 `__init__()`

A simplified variant of the Advanced Encryption Standard (AES).

Examples

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES(); maes
Mini-AES block cipher with 16-bit keys
sage: MS = MatrixSpace(FiniteField(16, "x"), 2, 2)
sage: P = MS.random_element()
sage: key = maes.random_key()
sage: C = maes.encrypt(P, key)
sage: plaintxt = maes.decrypt(C, key)
sage: plaintxt == P
True
```

D.3.2 __call__(B, key, algorithm="encrypt")

Apply Mini-AES encryption or decryption on the binary string `B` using the key `key`. The flag `algorithm` controls what action is to be performed on `B`.

Input

- `B` — a binary string, where the number of bits is positive and a multiple of 16. The number of 16 bits is evenly divided into four nibbles. Hence 16 bits can be conveniently represented as a 2×2 matrix block for encryption or decryption.

- `key` — a secret key; this must be a 16-bit binary string.

- `algorithm` — (default: `"encrypt"`) a string; a flag to signify whether encryption or decryption is to be applied to the binary string `B`. The encryption flag is `"encrypt"` and the decryption flag is `"decrypt"`.

Output

- The ciphertext (respectively plaintext) corresponding to the binary string `B`.

Examples

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: bin = BinaryStrings()
sage: key = bin.encoding("KE"); key
0100101101000101
sage: P = bin.encoding("Encrypt this secret message!"); P
010001010110111001100011011100100111100101110000011101000010000001110100011\
010000110100101110011001000000111001101100101011000110111001001100101011101\
000010000001101101011001010111001101110011011000010110011101110010100100001
sage: C = maes(P, key, algorithm="encrypt"); C
100010001010011011110000011110000100110011101101010001110110110101010010111\
011111010110011100111001000111011001010101000101001111101100110010100010001\
110110110100100000110001100011000001110000111001101011110000000001110001001
sage: plaintxt = maes(C, key, algorithm="decrypt")
sage: plaintxt == P
True
```

Exception tests

The binary string `B` must be non-empty and the number of bits must be a multiple of 16:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: maes = MiniAES()
sage: maes("B", "key")
Traceback (most recent call last):
...
TypeError: input B must be a non-empty binary string with number of\
bits a multiple of 16
sage: bin = BinaryStrings()
sage: B = bin.encoding("A")
sage: maes(B, "key")
Traceback (most recent call last):
...
ValueError: the number of bits in the binary string B must be positive\
and a multiple of 16
```

The secret key `key` must be a 16-bit binary string:

```
sage: B = bin.encoding("ABCD")
sage: maes(B, "key")
Traceback (most recent call last):
...
TypeError: secret key must be a 16-bit binary string
sage: key = bin.encoding("K")
sage: maes(B, key)
Traceback (most recent call last):
...
ValueError: secret key must be a 16-bit binary string
```

The value for `algorithm` must be either `"encrypt"` or `"decrypt"`:

```
sage: B = bin.encoding("ABCD")
sage: key = bin.encoding("KE")
sage: maes(B, key, algorithm="ABC")
Traceback (most recent call last):
...
ValueError: algorithm must be either 'encrypt' or 'decrypt'
sage: maes(B, key, algorithm="e")
Traceback (most recent call last):
...
ValueError: algorithm must be either 'encrypt' or 'decrypt'
sage: maes(B, key, algorithm="d")
Traceback (most recent call last):
...
ValueError: algorithm must be either 'encrypt' or 'decrypt'
```

D.3.3 `__eq__()`

Compare this Mini-AES cryptosystem object with the object represented by `other`.

Examples

Mini-AES objects are the same if they have the same key size and the same S-boxes:

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: m = MiniAES()
sage: m == loads(dumps(m))
True
```

D.3.4 __repr__()

Return the string representation of this Mini-AES cryptosystem.

Examples

```
sage: from sage.crypto.block_cipher.miniaes import MiniAES
sage: m = MiniAES(); m
Mini-AES block cipher with 16-bit keys
```

Appendix E

Sage Manual for Super-Increasing Sequences

Some basic functionalities for working with super-increasing sequences are implemented in the class

`sage.numerical.knapsack.Superincreasing` (E.1)

via bug tracking tickets `#6176` [81], `#5827` [79], and `#6222` [85]. This appendix provides the reference manual for the class (E.1). The three bug tracking tickets `#6176`, `#5827`, and `#6222` have been merged in the Sage standard library during the development of Sage versions 4.0.1.rc1 for the first two tickets, and version 4.0.2.alpha0 for the third ticket. The source code of the class (E.1) is available with the latest stable release of Sage, which as of this writing is Sage version 4.2.1.

E.1 Class documentation

A class for super-increasing sequences.

Let $L = (a_1, a_2, a_3, \ldots, a_n)$ be a non-empty sequence of non-negative integers. Then L is said to be super-increasing if each a_i is strictly greater than the sum of all previous values. That is, for each $a_i \in L$ the sequence L must satisfy the property

$$a_i > \sum_{k=1}^{i-1} a_k$$

in order to be called a super-increasing sequence, where $|L| \geq 2$. If L has only one element, it is also defined to be a super-increasing sequence.

If `seq` is `None`, then construct an empty sequence. By definition, this empty sequence is not super-increasing.

Input

- `seq` — (default: `None`) a non-empty sequence.

E.1.1 Example usage

Create a super-increasing sequence and perform some basic operations on it:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [1, 2, 5, 21, 69, 189, 376, 919]
sage: Superincreasing(L).is_superincreasing()
True
sage: Superincreasing().is_superincreasing([1,3,5,7])
False
sage: seq = Superincreasing(); seq
An empty sequence.
sage: seq = Superincreasing([1, 3, 6]); seq
Super-increasing sequence of length 3
sage: seq = Superincreasing(list([1, 2, 5, 21, 69, 189, 376, 919])); seq
Super-increasing sequence of length 8
```

E.2 Public methods

This section documents public methods in the class

`sage.numerical.knapsack.Superincreasing`

of the Sage standard library.

E.2.1 `is_superincreasing(seq=None)`

Determine whether or not `seq` is super-increasing.

If `seq=None` then determine whether or not self is super-increasing.

Let $L = (a_1, a_2, a_3, \ldots, a_n)$ be a non-empty sequence of non-negative integers. Then L is said to be super-increasing if each a_i is strictly greater than the sum of all previous values. That is, for each $a_i \in L$ the sequence L must satisfy the property

$$a_i > \sum_{k=1}^{i-1} a_k$$

in order to be called a super-increasing sequence, where $|L| \geq 2$. If L has exactly one element, then it is also defined to be a super-increasing sequence.

Input

- `seq` — (default: `None`) a sequence to test.

Output

- If `seq` is `None`, then test `self` to determine whether or not it is super-increasing. In that case, return `True` if self is super-increasing; `False` otherwise.
- If `seq` is not `None`, then test `seq` to determine whether or not it is super-increasing. Return `True` if `seq` is super-increasing; `False` otherwise.

Examples

By definition, an empty sequence is not super-increasing:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: Superincreasing().is_superincreasing([])
False
sage: Superincreasing().is_superincreasing()
False
sage: Superincreasing().is_superincreasing(tuple())
False
sage: Superincreasing().is_superincreasing(())
False
```

But here is an example of a super-increasing sequence:

```
sage: L = [1, 2, 5, 21, 69, 189, 376, 919]
sage: Superincreasing(L).is_superincreasing()
True
sage: L = (1, 2, 5, 21, 69, 189, 376, 919)
sage: Superincreasing(L).is_superincreasing()
True
```

A super-increasing sequence can have zero as one of its elements:

```
sage: L = [0, 1, 2, 4]
sage: Superincreasing(L).is_superincreasing()
True
```

A super-increasing sequence can be of length 1:

```
sage: Superincreasing([randint(0, 100)]).is_superincreasing()
True
```

Exception tests

The sequence must contain only integers:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [1.0, 2.1, pi, 21, 69, 189, 376, 919]
sage: Superincreasing(L).is_superincreasing()
...
TypeError: Element e (= 1.00000000000000) of seq must be a non-negative integer.
sage: L = [1, 2.1, pi, 21, 69, 189, 376, 919]
sage: Superincreasing(L).is_superincreasing()
...
TypeError: Element e (= 2.10000000000000) of seq must be a non-negative integer.
```

E.2.2 `largest_less_than(N)`

Return the largest integer in the sequence `self` that is less than or equal to `N`.

This function narrows down the candidate solution using a binary trim, similar to the way binary search halves the sequence at each iteration.

Input

- `N` — integer; the target value to search for.

Output

- The largest integer in `self` that is less than or equal to `N`. If no solution exists, then return `None`.

Examples

When a solution is found, return it:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [2, 3, 7, 25, 67, 179, 356, 819]
sage: Superincreasing(L).largest_less_than(207)
179
sage: L = (2, 3, 7, 25, 67, 179, 356, 819)
sage: Superincreasing(L).largest_less_than(2)
2
```

But if no solution exists, return `None`:

```
sage: L = [2, 3, 7, 25, 67, 179, 356, 819]
sage: Superincreasing(L).largest_less_than(-1) == None
True
```

Exception tests

The target `N` must be an integer:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [2, 3, 7, 25, 67, 179, 356, 819]
sage: Superincreasing(L).largest_less_than(2.30)
...
TypeError: N (= 2.30000000000000) must be an integer.
```

The sequence that `self` represents must also be non-empty:

```
sage: Superincreasing([]).largest_less_than(2)
...
ValueError: seq must be a super-increasing sequence
sage: Superincreasing(list()).largest_less_than(2)
...
ValueError: seq must be a super-increasing sequence
```

E.2.3 `subset_sum(N)`

Solving the subset sum problem for a super-increasing sequence.

Let $S = (s_1, s_2, s_3, \ldots, s_n)$ be a non-empty sequence of non-negative integers, and let $N \in \mathbf{Z}$ be non-negative. The subset sum problem asks for a subset $A \subseteq S$ all of whose elements sum to N. This method specializes the subset sum problem to the case of super-increasing sequences. If a solution exists, then it is also a super-increasing sequence.

Note

This method only solves the subset sum problem for super-increasing sequences. In general, solving the subset sum problem for an arbitrary sequence is known to be computationally hard.

Input

- `N` — a non-negative integer.

Output

- A non-empty subset of `self` whose elements sum to `N`. This subset is also a super-increasing sequence. If no such subset exists, then return the empty list.

Algorithm

The algorithm used is adapted from page 355 of Hoffstein et al. [48].

Examples

Solving the subset sum problem for a super-increasing sequence and target sum:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [1, 2, 5, 21, 69, 189, 376, 919]
sage: Superincreasing(L).subset_sum(98)
[69, 21, 5, 2, 1]
```

Exception tests

The target `N` must be a non-negative integer:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [0, 1, 2, 4]
sage: Superincreasing(L).subset_sum(-6)
...
TypeError: N (= -6) must be a non-negative integer.
sage: Superincreasing(L).subset_sum(-6.2)
...
TypeError: N (= -6.20000000000000) must be a non-negative integer.
```

The sequence that `self` represents must only contain non-negative integers:

```
sage: L = [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1]
sage: Superincreasing(L).subset_sum(1)
...
TypeError: Element e (= -10) of seq must be a non-negative integer.
```

E.3 Private methods

This section documents private methods implemented in the class

`sage.numerical.knapsack.Superincreasing`

of the Sage standard library.

E.3.1 `__init__(seq=None)`

Constructing a super-increasing sequence object from `seq`.

If `seq` is `None`, then construct an empty sequence. By definition, this empty sequence is not super-increasing.

Input

- `seq` — (default: `None`) a non-empty sequence.

Examples

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [1, 2, 5, 21, 69, 189, 376, 919]
sage: Superincreasing(L).is_superincreasing()
True
sage: Superincreasing().is_superincreasing([1,3,5,7])
False
```

E.3.2 `__cmp__(other)`

Comparing this super-increasing sequence object to that represented by `other`.

Exception tests

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [1, 2, 5, 21, 69, 189, 376, 919]
sage: seq = Superincreasing(L)
sage: seq == loads(dumps(seq))
True
```

E.3.3 `__repr__()`

Return a string representation of this super-increasing sequence object.

Examples

```
sage: from sage.numerical.knapsack import Superincreasing
sage: seq = Superincreasing(); seq
An empty sequence.
sage: seq = Superincreasing([1, 3, 6]); seq
Super-increasing sequence of length 3
sage: seq = Superincreasing(list([1, 2, 5, 21, 69, 189, 376, 919])); seq
Super-increasing sequence of length 8
```

E.3.4 _latex_()

Return LaTeX representation of this super-increasing sequence object.

Examples

```
sage: from sage.numerical.knapsack import Superincreasing
sage: latex(Superincreasing())
\left[\right]
sage: seq = Superincreasing([1, 2, 5, 21, 69, 189, 376, 919])
sage: latex(seq)
<BLANKLINE>
\left[1,
  2,
  5,
  21,
  69,
  189,
  376,
  919\right]
```

References

[1] Data Encryption Standard (DES). *Federal Information Processing Standard Publication 46*, 1977.

[2] Data Encryption Standard (DES). *Federal Information Processing Standard Publication 46-3*, 1999.

[3] Advanced Encryption Standard (AES). *Federal Information Processing Standard Publication 197*, 2001.

[4] ACM SIGCSE. Overview of the CS body of knowledge, 02nd November 2009. `http://www.sigcse.org/resources/cs-2001/al#AL-Cryptography`.

[5] M. Albrecht. Algebraic attacks on the Courtois toy cipher. Master's thesis, Department of Computer Science, Universität Bremen, Germany, 2006.

[6] M. Albrecht. Algebraic attacks on the Courtois toy cipher. *Cryptologia*, 32(3):220–276, 2008.

[7] M. Albrecht and C. Cid. Algebraic techniques in differential cryptanalysis. In O. Dunkelman, editor, *FSE 2009: Proceedings of the 16th International Workshop on Fast Software Encryption*, volume 5665 of *Lecture Notes in Computer Science*, pages 193–208. Springer, 2009.

[8] M. Albrecht, C. Gentry, S. Halevi, and J. Katz. Attacking cryptographic schemes based on "perturbation polynomials". Cryptology ePrint Archive, Report 2009/098, 2009. `http://eprint.iacr.org`.

[9] A. A. Aly and S. Akhtar. Cryptography and security protocols course for undergraduate IT students. *SIGCSE Bulletin*, 36(2):44–47, 2004.

[10] Y. Aner. Securing the Sage Notebook. Master's thesis, Royal Holloway, University of London, 2009.

[11] A. Baliga and S. Boztas. Cryptography in the classroom using Maple. In W. Yang, S. Chu, Z. Karian, and G. Fitz-Gerald, editors, *Proceedings of the Sixth Asian Technology Conference in Mathematics*, pages 343–350, 2001.

[12] G. V. Bard. *Algorithms for Solving Linear and Polynomial Systems of Equations over Finite Fields with Applications to Cryptanalysis.* PhD thesis, Department of Mathematics, University of Maryland, 2007.

[13] T. H. Barr. *Invitation to Cryptology.* Prentice Hall, 2002.

[14] R. A. Beezer. Sage (version 3.4). *SIAM Review*, 51(4):785–807, 2009.

[15] H. Beker and F. Piper. *Cipher Systems: The Protection of Communications.* John Wiley and Sons, 1982.

[16] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters. Twisted Edwards curves. In S. Vaudenay, editor, *AFRICACRYPT 2008: First International Conference on Cryptology in Africa*, volume 5023 of *Lecture Notes in Computer Science*, pages 389–405. Springer, 2008.

[17] D. J. Bernstein, T. Lange, and R. R. Farashahi. Binary Edwards curves. In E. Oswald and P. Rohatgi, editors, *CHES '08: Proceedings of the 10th International Workshop on Cryptographic Hardware and Embedded Systems*, volume 5154 of *Lecture Notes in Computer Science*, pages 244–265. Springer, 2008.

[18] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. *Keccak Sponge Function Family: Main Document, Version 2.0*, 24th October 2009. `http://keccak.noekeon.org/Keccak-main-2.0.pdf`.

[19] D. Bishop. *Introduction to Cryptography with Java Applets*. Jones and Bartlett Publishers, 2003.

[20] D. Boneh, C. Gentry, and M. Hamburg. Space-efficient identity based encryption without pairings. In *FOCS '07: Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, pages 647–657. IEEE Computer Society, 2007.

[21] W. Bosma, J. Cannon, and C. Playoust. The MAGMA algebra system I: the user language. *Journal of Symbolic Computation*, 24(3-4):235–265, 1997.

[22] B. Buchanan. Math 478 - cryptography, 04th November 2009. `http://banach.millersville.edu/~bob/math478/`.

[23] B. Buchberger. Should students learn integration rules? *ACM SIGSAM Bulletin*, 24(1):10–17, 1990.

[24] B. Buchberger. Computer algebra: The end of mathematics? *ACM SIGSAM Bulletin*, 36(1):3–9, 2002.

[25] K. W. Campbell and M. J. Wiener. DES is not a group. In E. F. Brickell, editor, *CRYPTO '92—Proceedings of the 12th Annual International Cryptology Conference*, volume 740 of *Lecture Notes in Computer Science*, pages 512–520. Springer, 1992.

[26] S. K. Chong, G. Farr, L. Frost, and S. Hawley. On pedagogically sound examples in public-key cryptography. In V. Estivill-Castro and G. Dobbie, editors, *Twenty-Nineth Australasian Computer Science Conference (ACSC2006)*, volume 48 of *CRPIT*, pages 63–68. Australian Computer Society, 2006.

[27] C. Cid, S. Murphy, and M. Robshaw. *Algebraic Aspects of the Advanced Encryption Standard*. Springer, 2006.

[28] C. Cid, S. Murphy, and M. J. B. Robshaw. Small scale variants of the AES. In H. Gilbert and H. Handschuh, editors, *Proceedings of the 12th International Workshop on Fast Software Encryption*, volume 3557 of *Lecture Notes in Computer Science*, pages 145–162. Springer, 2005.

[29] H. Cohen et al. *Pari/GP Computer Algebra System (Version 2.3.3)*. The PARI Group, 24th October 2009. `http://pari.math.u-bordeaux.fr`.

[30] J. Cosgrave. Number theory and cryptography (using Maple). In D. Joyner, editor, *Coding Theory and Cryptography: From Enigma to Geheimschreiber to Quantum Theory*, pages 124–143. Springer, 2000.

[31] N. T. Courtois. How fast can be algebraic attacks on block ciphers? In E. Biham, H. Handschuh, S. Lucks, and V. Rijmen, editors, *Symmetric Cryptography*, number 07021 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[32] M. Curtin. *Brute Force: Cracking The Data Encryption Standard*. Copernicus Books, 2005.

[33] J. Daemen and V. Rijmen. *The Design of Rijndael: AES — The Advanced Encryption Standard*. Springer, 2002.

[34] P. N. de Souza, R. J. Fateman, J. Moses, and C. Yapp. The Maxima book, 30th October 2009. `http://maxima.sourceforge.net/docs/maximabook/maximabook-19-Sept-2004.pdf`.

[35] M. Eisenberg. Hill ciphers: A linear algebra project with Mathematica. In G. Goodell, editor, *Proceedings of the Twelfth Annual International Conference on Technology in Collegiate Mathematics*, pages 100–104. Addison-Wesley, 2001.

[36] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31(4):469–472, 1985.

[37] B. Esslinger et al. *CrypTool: Cryptography Educational Tool (Version 1.4.21)*. The CrypTool Development Team, 24th October 2009. `http://www.cryptool.org`.

[38] N. Ferguson and B. Schneier. *Practical Cryptography*. Wiley Publishing, 2003.

[39] Free Software Foundation. GNU General Public License Version 2, 04th November 2009. `http://www.gnu.org/licenses/old-licenses/gpl-2.0.html`.

[40] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4.12*, 2008. `http://www.gap-system.org`.

[41] S. Garera and J. Vasconcelos. Challenges in teaching a graduate course in applied cryptography. *SIGCSE Bulletin*, 41(2):103–107, 2009.

[42] T. Gautier, J.-L. Roch, G. Villard, J.-G. Dumas, P. Giorgi, and C. Pernet. *Givaro: C++ library for arithmetic and algebraic computations*, 05th November 2009. `http://ljk.imag.fr/CASYS/LOGICIELS/givaro`.

[43] M. Gray. Sage: A new mathematics software system. *Computing in Science & Engineering*, 10(6):72–75, Nov.-Dec. 2008.

[44] W. B. Hart et al. *MPIR: Multiple Precision Integers and Rationals*, 05th November 2009. `http://www.mpir.org`.

[45] A. C. Hearn. *REDUCE: A portable general-purpose computer algebra system*, 02nd November 2009. `http://reduce-algebra.sourceforge.net`.

[46] L. S. Hill. Cryptography in an algebraic alphabet. *The American Mathematical Monthly*, 36(6):306–312, 1929.

[47] L. S. Hill. Concerning certain linear transformation apparatus of cryptography. *The American Mathematical Monthly*, 38(3):135–154, 1931.

[48] J. Hoffstein, J. Pipher, and J. H. Silverman. *An Introduction to Mathematical Cryptography*. Springer, 2008.

[49] T. W. Hungerford. *Abstract Algebra: An Introduction*. Thomson Learning, 2nd edition, 1997.

[50] D. Joyner. Open source computer algebra systems: Maxima. *ACM Communications in Computer Algebra*, 40(3):92–96, 2006.

[51] D. Joyner and W. Stein. Open source mathematical software. *Notices of the American Mathematical Society*, 54(10):1279, 2007.

[52] R. E. Klima, N. P. Sigmon, and E. L. Stitzinger. *Applications of Abstract Algebra with Maple and Matlab*. Chapman & Hall/CRC, 2nd edition, 2007.

[53] N. Koblitz. *Algebraic Aspects of Cryptography*. Springer, 2004.

[54] W. Koepf. Mathematics with DERIVE as didactical tool. *The DERIVE-Newsletter*, 38:23–35, 2000.

[55] D. R. Kohel. Cryptography, 05th November 2009. `http://echidna.maths.usyd.edu.au/~kohel/tch/Crypto/crypto.pdf`.

[56] A. M. Kuchling. *Python Cryptography Toolkit (Version 2.0.1)*, 04th December 2009. `http://www.amk.ca/python/code/crypto`.

[57] R. E. Lewand. *Cryptological Mathematics*. The Mathematical Association of America, 2000.

[58] R. Lidl and H. Niederreiter. *Finite Fields*. Cambridge University Press, 2nd edition, 1997.

[59] S. Maitra and S. Sarkar. A new class of weak encryption exponents in RSA. In D. R. Chowdhury, V. Rijmen, and A. Das, editors, *Progress in Cryptology – INDOCRYPT 2008, 9th International Conference on Cryptology*, volume 5365 of *Lecture Notes in Computer Science*, pages 337–349. Springer, 2008.

[60] S. Maitra and S. Sarkar. Revisiting Wiener's attack — new weak keys in RSA. In Tzong-ChenWu, C.-L. Lei, V. Rijmen, and D.-T. Lee, editors, *ISC '08: Proceedings of the 11th International Conference on Information Security*, volume 5222 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2008.

[61] W. A. Martin and R. J. Fateman. The MACSYMA system. In *SYMSAC '71: Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, pages 59–75. ACM, 1971.

[62] Maxima.sourceforge.net. *Maxima, a Computer Algebra System, Version 5.19.1*, 02nd November 2009. `http://maxima.sourceforge.net`.

[63] M. May. Maple worksheets for cryptography, 27th October 2009. `http://euler.slu.edu/courseware/CryptoSubmissionSet/Cryptography.html`.

[64] M. May. Using Maple worksheets to enable student explorations of cryptography. *Cryptologia*, 33(2):151–157, 2009.

[65] A. McAndrew. Computer algebra systems: An introductory view. Technical Report 18MATH2, Victoria University of Technology, January 1992.

[66] A. McAndrew. Teaching cryptography with open-source software. In J. D. Dougherty, S. H. Rodger, S. Fitzgerald, and M. Guzdial, editors, *SIGCSE 2008: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, pages 325–329. Association for Computing Machinery, 2008.

[67] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[68] R. A. Mollin. *An Introduction to Cryptography*. Chapman & Hall/CRC, 2nd edition, 2007.

[69] M. B. Monagan. 2D and 3D graphical routines for teaching linear algebra. *Proceedings of the 2002 Maple Summer Workshop*, 2002.

[70] G. Moody. *Rebel Code: Linux and the Open Source Revolution*. Basic Books, 2001.

[71] V. Muñoz and U. Persson. Interviews with three Fields medalists. *Notices of the American Mathematical Society*, 54(3):405–410, 2007.

[72] M. A. Musa, E. F. Schaefer, and S. Wedig. A simplified AES algorithm and its linear and differential cryptanalysis. *Cryptologia*, 27(2):148–177, 2003.

[73] L. Naismith and C. J. Sangwin. Computer algebra based assessment of mathematics online. In *Proceedings of the 8th Computer-Assisted Assessment Conference*, Loughborough, UK, 2004. Loughborough University.

[74] J. Neubüser. An invitation to computational group theory. In C. M. Campbell, T. C. Hurley, E. F. Robertson, S. J. Tobin, and J. J. Ward, editors, *Groups '93 Galway/St. Andrews, Volume 2*, volume 212 of *London Mathematical Society Lecture Note Series*, pages 457–475. Cambridge University Press, 1995.

[75] W. Neun. REDUCE is free software as of January 2009. *ACM Communications in Computer Algebra*, 43(1):15–16, 2009.

[76] M. V. Nguyen. Affine cipher and its cryptanalysis, 2009. `http://trac.sagemath.org/sage_trac/ticket/7124`.

[77] M. V. Nguyen. Bring documentation of classical.py to 100%, 2009. `http://trac.sagemath.org/sage_trac/ticket/5529`.

[78] M. V. Nguyen. Cryptanalysis of the shift cipher, 2009. `http://trac.sagemath.org/sage_trac/ticket/7123`.

[79] M. V. Nguyen. Knapsack: subset sum problem for super-increasing sequences, 2009. `http://trac.sagemath.org/sage_trac/ticket/5827`.

[80] M. V. Nguyen. Phan's mini-aes for educational purposes, 2009. `http://trac.sagemath.org/sage_trac/ticket/6164`.

[81] M. V. Nguyen. Restify modules in sage/numerical and put them in reference manual, 2009. `http://trac.sagemath.org/sage_trac/ticket/6176`.

[82] M. V. Nguyen. Sanity check key value of the shift cryptosystem, 2009. `http://trac.sagemath.org/sage_trac/ticket/7010`.

[83] M. V. Nguyen. Schaefer's simplified data encryption standard for educational purposes, 2009. `http://trac.sagemath.org/sage_trac/ticket/6461`.

[84] M. V. Nguyen. The shift cryptosystem, 2009. `http://trac.sagemath.org/sage_trac/ticket/6841`.

[85] M. V. Nguyen. Typos in super-increasing code, 2009. `http://trac.sagemath.org/sage_trac/ticket/6222`.

[86] NIST. Announcing approval of the withdrawal of federal information processing standard (FIPS) 46-3, data encryption standard (DES); FIPS 74, guidelines for implementing and using the NBS data encryption standard; and FIPS 81, DES modes of operation. *Federal Register*, 70(96):28907–28908, 2005.

[87] NIST. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register*, 72(212):62212–62220, 2007.

[88] A. Nitaj. Another generalization of Wiener's attack on RSA. In S. Vaudenay, editor, *AFRICACRYPT 2008: Proceedings of the First International Conference on Cryptology in Africa*, volume 5023 of *Lecture Notes in Computer Science*, pages 174–190. Springer, 2008.

[89] S. Ockman, M. Stone, and C. Dibona, editors. *Open Sources: Voices from the Open Source Revolution*. O'Reilly Media, 1999.

[90] OSI. Open Source Initiative, 02nd November 2009. `http://www.opensource.org`.

[91] P. D. Palma, C. Frank, S. E. Gladfelter, and J. Holden. Cryptography and computer security for undergraduates. In D. Joyce, D. Knox, W. Dann, and T. L. Naps, editors, *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pages 94–95. Association for Computing Machinery, 2004.

[92] W. Patterson. *Mathematical Cryptology for Computer Scientists and Mathematicians*. Rowman & Littlefield, 1987.

[93] R. C.-W. Phan. Mini advanced encryption standard (Mini-AES): A testbed for cryptanalysis students. *Cryptologia*, 26(4):283–306, 2002.

[94] A. Z. Pinkus and S. Winitzki. YACAS: A do-it-yourself symbolic algebra environment. In J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, editors, *AISC 2002: Proceedings of the Joint International Conferences on Artificial Intelligence, Automated Reasoning, and Symbolic Computation*, volume 2385 of *Lecture Notes in Computer Science*, pages 332–336. Springer, 2002.

[95] B. Pletsch. Computer algebra in mathematics education. In M. J. Wester, editor, *Computer Algebra Systems: A Practical Guide*, pages 285–322. John Wiley & Sons, 1999.

[96] M. O. Rabin. Digitized signatures and public-key functions as intractable as factorization. Technical Report LCS/TR-212, Massachusetts Institute of Technology, 1979.

[97] E. S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Media, 2001.

[98] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[99] K. H. Rosen. *Discrete Mathematics and Its Applications*. WCB/McGraw-Hill, 4th edition, 1999.

[100] A. D. Rubin. An experience teaching a graduate course in cryptography. *Cryptologia*, 21(2):97–109, 1997.

[101] sage.math. The `sage.math` compute node, 18th October 2009. `http://sage.math.washington.edu`.

[102] E. F. Schaefer. A simplified data encryption standard algorithm. *Cryptologia*, 20(1):77–84, 1996.

[103] R. Schlesinger. A cryptography course for non-mathematicians. In *InfoSecCD '04: Proceedings of the 1st annual conference on Information security curriculum development*, pages 94–98. Association for Computing Machinery, 2004.

[104] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 2nd edition, 1996.

[105] B. Schneier. *Secrets & Lies: Digital Security in a Networked World*. Wiley Publishing, 2000.

[106] V. Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2nd edition, 2009.

[107] V. Shoup. *NTL: A Library for doing Number Theory*, 05th November 2009. `http://www.shoup.net/ntl`.

[108] R. Spillman. A software tool for teaching classical & contemporary cryptology. *Journal of Computing Sciences in Colleges*, 20(2):114–124, 2004.

[109] R. M. Stallman, L. Lessig, and J. Gay. *Free Software, Free Society: Selected Essays of Richard M. Stallman*. Free Software Foundation, 2002.

[110] W. Stein. Can we create a viable free open source alternative to Magma, Maple, Mathematica and Matlab? In J. R. Sendra and L. González-Vega, editors, *ISSAC 2008: Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 5–6. Association for Computing Machinery, 2008.

[111] W. Stein et al. *Sage Mathematics Software (Version 4.2.1)*. The Sage Development Team, 14th November 2009. `http://www.sagemath.org`.

[112] W. Stein and D. Joyner. SAGE: System for algebra and geometry experimentation. *ACM SIGSAM Bulletin*, 39(2):61–64, 2005.

[113] D. R. Stinson. *Cryptography: Theory and Practice*. Chapman & Hall/CRC, 3rd edition, 2006.

[114] W. Trappe and L. C. Washington. *Introduction to Cryptography with Coding Theory*. Prentice Hall, 2nd edition, 2006.

[115] W. Trappe and L. C. Washington. Introduction to cryptography with coding theory, 2nd edition, book website, 26th October 2009. `http://www-users.math.umd.edu/~lcw/book.html`.

[116] V. Velichkov, V. Rijmen, and B. Preneel. Algebraic cryptanalysis of a small-scale version of stream cipher LEX. In *Proceedings of the 30th Symposium on Information Theory in the Benelux*, 2009.

[117] J. E. Villate. Teaching dynamical systems with Maxima. In *ACA 2007: Proceedings of the 2007 International Conference on Applications of Computer Algebra*, Rochester, MI, USA, 2007. Oakland University.

[118] R.-P. Weinmann. *Algebraic Methods in Block Cipher Cryptanalysis*. PhD thesis, Department of Computer Science, Technischen Universität Darmstadt, Germany, 2009.

[119] M. J. Wiener. Cryptanalysis of short RSA secret exponents. *IEEE Transactions on Information Theory*, 36(3):553–558, 1990.

[120] S. Williams. *Free as in Freedom: Richard Stallman's Crusade for Free Software*. O'Reilly Media, 2002.

[121] S. Y. Yan. *Number Theory for Computing*. Springer, 2nd edition, 2002.