Lightweight Introduction to Lattices

(Miroslav Dimitrov, Harald Ritter, Bernhard Esslinger, April 2020, v.1.0.6)

1 Introduction

In the following sections, our goal is to cover the basic theory behind lattices in a lightweight fashion. The covered theory will be accompanied with lots of practical examples, *SageMath* code, and cryptographic challenges.

Sections 2 to 8 introduce the notation and methods needed to deal with and understand lattices (this makes up around one third of this chapter). Sections 9 to 10 deal in more detail with lattices and their application to attack RSA. Section 11 is considered as a deeper outlook providing some algorithms for lattice basis reduction and their usage to break cryptosystems. The appendix contains screenshots where to find lattice algorithms in the CrypTool programs.

2 Preliminaries

You are not required to have any advanced background in any mathematical domain or programming language. Nevertheless, expanding your knowledge and learning new mathematical concepts and programming techniques will give you a great boost towards your goal as a future expert in cryptology. This chapter is built in an independent way – you are not required to read the previous chapters in the book. The examples and the practical scenarios will be implemented in *SageMath* (a computer-algebra system (CAS), which uses Python as scripting language). We believe that the code in this chapter is very intuitive and even if you don't have any experience with Python syntax you will grasp the idea behind it. However, if you want to learn Python or just polish up your current skills we recommend the freely available book of Charles Severance[1].

[1] Charles Severance: *Python for Informatics* from `http://www.pythonlearn.com/`, Version 2.7.3, 2015. At `www.py4e.com`, there is also a version about Python 3 from Charles Severance called *Python for Everybody*. From version v9.0 of the open-source CAS *SageMath* it is based on Python 3.

3 Equations

According to an English dictionary[2] "equation" means "statement that the values of two mathematical expressions are equal (indicated by the sign =)".

The equations could be trivial (without an unknown variable involved):

$$\begin{aligned} 0 &= 0 \\ 1 &= 1 \\ 1+1 &= 2 \\ 1.\overline{9} &= 2 \end{aligned}$$

... or with variables (also called indeterminates) which leads to a set of solutions or "the" one solution or no solution (unsolvable):

$$\begin{aligned} x+x &= 10 \\ x+y &= 10 \\ x+y &= z \\ x_1+x_2+x_3+\cdots+x_{10} &= z \end{aligned}$$

Equations help us to mathematically describe a relationship between some objects. In some cases, the solution is straightforward and unique, but in some other cases we have a set of possible solutions. By **domain** we define the set of input values for which the equation is defined. As example, the equation $x+1=10$ has one solution over the positive integer numbers and no solutions over the negative integer numbers. From now on till the end of this chapter we are going to work only with the set of integer numbers $\mathbb{Z}$.

In *SageMath* we can easily define variables. The following declaration defines the special symbol x as a variable:

```
sage: x = var('x')
```

Now, we can construct our equation. Let's say that we want to find the solution of the following equation $x+x^2+x^3=100$. First, we need to define our left side of the equation. The declaration using *SageMath* is straightforward. We will name the left side of our equation as **leq**.

Here is an example[3] with coefficients (they need the times operator $*$).

[2] https://en.oxforddictionaries.com/definition/equation
Equations are mathematical statements which might be wrong or right. E.g. $1.\overline{9}=2$ is right despite many people at first don't think so. But: $0.\overline{3}=\frac{1}{3} \Rightarrow 0.\overline{9}=3\cdot 0.\overline{3}=3\cdot\frac{1}{3}=1$

[3] The symbols $**$ mean in both *SageMath* and Python powered to.
The expression on the right side is called a polynomial. A **polynomial** P consists of variables and coefficients, that involve only the operations of addition, subtraction, multiplication, and non-negative

```
sage: leq1 = x + 2*x**2 + x**3
```

For our example we use a term without coefficients:

```
sage: leq2 = x + x**2 + x**3
```

We are ready to solve the equation and find the solutions.

```
sage: eq_sol = solve(leq2==100, x)
```

The last command of *SageMath* will try to find all x for which $x + x^2 + x^3 = 100$. If you try this on your machine, you will notice a list of possible solutions which don't look like integer numbers. This is, because we defined x as variable without restrictions of its domain. So, let's predefine the symbol x as variable in the domain of integer numbers:[4]

```
sage: x = var('x', domain=ZZ)
```

Now, when we try to solve the equation we receive as list of solutions the empty list[5]. This means that there is no such x which satisfies the defined equation. And indeed, let's see the values of the equation for consecutive values of x.

```
sage: for i in range(-6,6):
....:        print(leq2(x=i), "for", "x=", i)
-186  for x= -6
-105  for x= -5
 -52  for x= -4
 -21  for x= -3
  -6  for x= -2
  -1  for x= -1
   0  for x= 0
   3  for x= 1
  14  for x= 2
  39  for x= 3
  84  for x= 4
 155  for x= 5
 258  for x= 6
```

We can notice two characteristics of our equation. First, the left side of the equation becomes larger when the variable x grows. Second, the solution for our equation is some non-integer number between 4 and 5.

integer exponents of variables. An example of a polynomial of a single variable x is $x^2 - 4x + 7$. The highest exponent in the variable is called the degree of the polynomial. Every polynomial P in a single variable x defines a function $x \to P(x)$, and is also called a univariate polynomial. An integer polynomial allows its variables and coefficients to be only integer values.
See `https://en.wikipedia.org/wiki/Polynomial`.

[4] **ZZ** means the domain of integer numbers $\mathbb{Z}$. Integers can be further limited to the Boolean type (true or false) or the integers modulo n (IntegerModRing(n) or GF(n), if n is prime).

[5] [] defines an empty list.

Remark: Normally, the term on the left contains also the number on the right side, if it's different from 0. So the usual way to write this equation is: $x^3 + x^2 + x - 100 = 0$.

Challenge 1 *We found this strange list of (independent) equations. Can you find the hidden message? Consider the found values for the variable of each equation as the ascii value of a character. The correct ascii values build the correct word in the same order as the 8 equations appear here. A copy-and-paste-ready version of this equation system is also available, see page 75.*

$$\begin{aligned}
x_0^4 - 150x_0^3 + 4389\,x_0^2 - 43000\,x_0 + 131100 &= 0\\
x_1^{10} - 177\,x_1^9 + 9143\,x_1^8 - 228909\,x_1^7 + 3264597\,x_1^6 - 28298835\,x_1^5 + &\\
+152170893\,x_1^4 - 502513551\,x_1^3 + 974729862\,x_1^2 - 995312448\,x_1 + 396179424 &= 0\\
x_2^{10} - 196\,x_2^9 + 12537\,x_2^8 - 397764\,x_2^7 + 7189071\,x_2^6 - 77789724\,x_2^5 + &\\
+506733203\,x_2^4 - 1941451916\,x_2^3 + 4165661988\,x_2^2 - 4501832400\,x_2 + 1841875200 &= 0\\
x_3^5 - 153\,x_3^4 + 5317\,x_3^3 - 77199\,x_3^2 + 510274\,x_3 - 1269840 &= 0\\
x_4^8 - 194\,x_4^7 + 11791\,x_4^6 - 352754\,x_4^5 + 6011644\,x_4^4 - &\\
-61295576\,x_4^3 + 370272864\,x_4^2 - 1222050816\,x_4 + 1696757760 &= 0\\
x_5^6 - 169\,x_5^5 + 7702\,x_5^4 - 153082\,x_5^3 + 1477573\,x_5^2 - 6672349\,x_5 + 11042724 &= 0\\
x_6^8 - 202\,x_6^7 + 12936\,x_6^6 - 406082\,x_6^5 + 7170059\,x_6^4 - &\\
-74124708\,x_6^3 + 439747164\,x_6^2 - 1365683328\,x_6 + 1701311040 &= 0\\
x_7^9 - 206\,x_7^8 + 13919\,x_7^7 - 467924\,x_7^6 + 8975099\,x_7^5 - 102829454\,x_7^4 + &\\
+699732361\,x_7^3 - 2673468816\,x_7^2 + 4956440220\,x_7 - 2888395200 &= 0
\end{aligned}$$

4 System of Linear Equations

We already introduced the concepts of variables, equations and the domain of an equation. We showed how to declare variables in *SageMath* and how to find solutions of single-variable equations automatically with *solve*. What if we have two different variables in our equation? Let's take as an example the following equation $x + y = 10$. Let's try to solve this equation using *SageMath*:[6]

```
sage: x = var('x', domain=ZZ)
sage: y = var('y', domain=ZZ)
sage: solve(x+y==10, (x,y))
(t_0, -t_0 + 10)
```

[6]This time we require as solution of `solve()` the tuple (x, y).

We get as solution $x = t_0$ and $y = -t_0 + 10$ and indeed $x + y = t_0 + (-t_0 + 10) = t_0 - t_0 + 10 = 10$. This notation is used by *SageMath* to show us that there are infinite many integer solutions to the given equation.

What if we have another restrictions of x and y. As example, what if we know that they are equal? Equality defines another equation which is related to the first one, so we can form a system of equations (in a **system of equations** the single equations are not independent from each other):

$$\begin{cases} x + y = 10 \\ x = y \end{cases}$$

Let's solve this system of equations using *SageMath*. We can easily organize all the equations from this 2-equation system using a list array.

```
sage: x = var('x', domain=ZZ)
sage: y = var('y', domain=ZZ)
sage: solve([x+y==10,x==y], (x,y))
[[x == 5, y == 5]]
```

As we see, we have only one solution $x = y = 5$.

A rich collection of mathematical problems can be solved by using systems of linear equations. Let's take as example the simple puzzle in fig. 4.1 and solve it with *SageMath*.

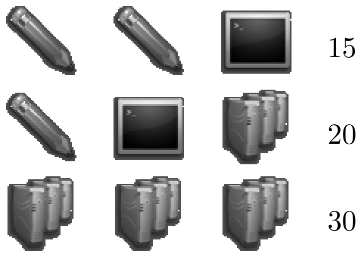


Figure 4.1: Visual Puzzle

As usual, each row consists of three different items and their corresponding total price. Usually, the goal in such puzzles is to find the price of each distinct item. We have 3 distinct items. Let's define the price of each pencil as x, the price of each computer display as y and the price of each bundle of servers as z. Following the previous declarations we can write down the following system of linear equations:

$$\begin{cases} 2x + y = 15 \\ x + y + z = 20 \\ 3z = 30 \end{cases}$$

We can easily solve this puzzle by using pen and paper only. The last equation reveals the value of $z = 10$. Eliminating the variable z by replacing its value in the previous equations, we reduce the system to system of two unknown variables:

$$\begin{cases} 2x + y = 15 \\ x + y = 10 \\ z = 10 \end{cases}$$

We can now subtract the second equation from the first one to receive:

$$\begin{aligned} 2x + y - (x + y) = 15 - 10 = 5 \\ x = 5 \end{aligned}$$

Therefore, we ended up with the following solution of the puzzle:

$$\begin{cases} x = 5 \\ y = 5 \\ z = 10 \end{cases}$$

Now, let's try to solve the same puzzle by using *SageMath*.

```
sage: x = var('x', domain=ZZ)
sage: y = var('y', domain=ZZ)
sage: z = var('z', domain=ZZ)
sage: solve([x + x + y == 15, x+y+z == 20, z+z+z == 30],
      (x,y,z))
[[x == 5, y == 5, z == 10]]
```

Challenge 2 *Can you recover the hidden message in the picture puzzle in fig. 4.2 on the next page? Each symbol represents a distinct decimal digit. There is a balance that each left side equals the corresponding right side. Automate the process by using* SageMath*. Hint: ASCII (American Standard Code for Information Interchange) is involved.*

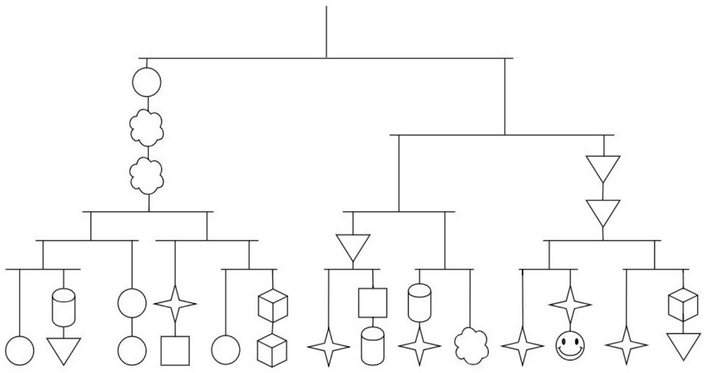


Figure 4.2: Puzzle Challenge (picture created by the author)

In the next sections we are going to introduce the definition of matrices, which will help us describe a given system of linear equations in a much more compact way.

5 Matrices

Is there more convenient way to write down huge systems of linear equations? We are going to introduce this way by using augmented matrices. You can consider a matrix as a rectangular or square array of numbers. The numbers are arranged in rows and columns. For example, let's analyze the following matrix:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \end{bmatrix}$$

We have 2 rows and 3 columns, and a total of 6 elements. We define an element as $a_{i,j}$ when we want to emphasize that the element is located on the i-th row and j-th column[7]. For example, $a_{1,1} = 1$, $a_{1,3} = 3$, $a_{2,2} = 4$.

In the following system of linear equations the independent variables are called a, b, c, d, e, and f.

$$\begin{cases} 6a + 7b + 11c + 18d + 4e + 7f = 5 \\ 8a + 14b + 2c + 13d + 2e + f = 19 \\ a + b + 3c + 4d + 4e + 7f = 15 \\ 3a + 4b + c + d + 14e + 17f = 1 \\ 5a + 5b + 2c + 2d + 2e + 6f = 2 \\ 11a + 17b + c + d + e + f = 9 \end{cases}$$

[7]Here we use the indexing starting from 1 as usual in mathematics. However, later in the *SageMath* samples the index of row and column start from 0 (as usual in computer languages like C or Python).

We can easily write down this system of linear equations as a matrix. Let's write down all coefficients in front of the variable a in the first column of our new matrix, all coefficients in front of the variable b in the second column and so on. This forms the **coefficient matrix**.

The right side of each equation forms another column – the last one. For clarity we are going to separate it from other columns with a vertical line. We call such matrix an **augmented matrix**.

$$\left(\begin{array}{cccccc|c} 6 & 7 & 11 & 18 & 4 & 7 & 5 \\ 8 & 14 & 2 & 13 & 2 & 1 & 19 \\ 1 & 1 & 3 & 4 & 4 & 7 & 15 \\ 3 & 4 & 1 & 1 & 14 & 17 & 1 \\ 5 & 5 & 2 & 2 & 2 & 6 & 2 \\ 11 & 17 & 1 & 1 & 1 & 1 & 9 \end{array}\right)$$

Let's analyze the behavior of a system of linear equations. We can make the following observations:

- Swapping the positions of two equations doesn't affect the solution of the system of the linear equations.
- Multiplying the equation by a nonzero number doesn't affect the solution of the system of the linear equations.
- Adding one randomly chosen equation to another randomly chosen equation doesn't affect the solution of the system of the linear equations.

We can easily transform those properties as properties in the augmented matrix. Furthermore, those properties allow us to build a complete automatic system of finding solutions to a given augmented matrix. In linear algebra, **Gaussian elimination** (also known as row reduction) is an algorithm for solving systems of linear equations. It is usually understood as a sequence of operations performed on the corresponding augmented matrix. Once all of the leading coefficients (the leftmost nonzero entry in each row) are 1, and every column containing a leading coefficient has zeros elsewhere, the matrix is said to be in **reduced row echelon form**.[8]

Let's apply Gaussian elimination on the following system of linear equations:

$$\begin{cases} 4x + 8y + 3z = 10 \\ 5x + 6y + 2z = 15 \\ 9x + 5y + z = 20 \end{cases}$$

[8]See https://en.wikipedia.org/wiki/Row_echelon_form

We can easily transform this system of linear equations to an augmented matrix.

$$\left(\begin{array}{ccc|c} 4 & 8 & 3 & 10 \\ 5 & 6 & 2 & 15 \\ 9 & 5 & 1 & 20 \end{array}\right)$$

Then, we start to transform the matrix to row echelon form. We first divide the first row by 4.

$$\left(\begin{array}{ccc|c} 4 & 8 & 3 & 10 \\ 5 & 6 & 2 & 15 \\ 9 & 5 & 1 & 20 \end{array}\right) \to \left(\begin{array}{ccr|c} 1 & 2 & 0.75 & 2.5 \\ 5 & 6 & 2 & 15 \\ 9 & 5 & 1 & 20 \end{array}\right)$$

The reason for dividing the first row by 4 is simple – we need the first element of the first row to be equal to 1, which allows us to multiply the first row by consequently 5 and 9 and to subtract it from respectively the second and the third row. Let's recall that we are trying to transform the augmented matrix to reduced row echelon form. Now, let's apply the previously made observations.

$$\left(\begin{array}{ccr|c} 1 & 2 & 0.75 & 2.5 \\ 5 & 6 & 2 & 15 \\ 9 & 5 & 1 & 20 \end{array}\right) \to \left(\begin{array}{crr|c} 1 & 2 & 0.75 & 2.5 \\ 0 & -4 & -1.75 & 2.5 \\ 9 & 5 & 1 & 20 \end{array}\right) \to \left(\begin{array}{crr|r} 1 & 2 & 0.75 & 2.5 \\ 0 & -4 & -1.75 & 2.5 \\ 0 & -13 & -5.75 & -2.5 \end{array}\right)$$

Now, we divide the second row with -4. This will transform the second element of the second row to 1 and will allow us to continue with our strategy of reducing the augmented matrix to the required row echelon form.

$$\left(\begin{array}{crr|r} 1 & 2 & 0.75 & 2.5 \\ 0 & -4 & -1.75 & 2.5 \\ 0 & -13 & -5.75 & -2.5 \end{array}\right) \to \left(\begin{array}{crr|r} 1 & 2 & 0.75 & 2.5 \\ 0 & 1 & 0.4375 & -0.625 \\ 0 & -13 & -5.75 & -2.5 \end{array}\right)$$

And again following the previous strategy we applied on the first row, we multiply the second row by 2 and subtracts it from the first row. Right after this operation we multiply the second row by 13 and add it to the last row.

$$\left(\begin{array}{crr|r} 1 & 2 & 0.75 & 2.5 \\ 0 & 1 & 0.4375 & -0.625 \\ 0 & -13 & -5.75 & -2.5 \end{array}\right) \to \left(\begin{array}{crr|r} 1 & 0 & -0.125 & 3.75 \\ 0 & 1 & 0.4375 & -0.625 \\ 0 & -13 & -5.75 & -2.5 \end{array}\right)$$

$$\left(\begin{array}{crr|r} 1 & 0 & -0.125 & 3.75 \\ 0 & 1 & 0.4375 & -0.625 \\ 0 & -13 & -5.75 & -2.5 \end{array}\right) \to \left(\begin{array}{ccr|r} 1 & 0 & -0.125 & 3.75 \\ 0 & 1 & 0.4375 & -0.625 \\ 0 & 0 & -0.0625 & -10.625 \end{array}\right)$$

We are almost ready. Now, we normalize the last row by dividing it by -0.0625.

$$\left(\begin{array}{ccc|c} 1 & 0 & -0.125 & 3.75 \\ 0 & 1 & 0.4375 & -0.625 \\ 0 & 0 & -0.0625 & -10.625 \end{array}\right) \to \left(\begin{array}{ccc|c} 1 & 0 & -0.125 & 3.75 \\ 0 & 1 & 0.4375 & -0.625 \\ 0 & 0 & 1 & 170 \end{array}\right)$$

We are following the same steps as the previously applied operations. First, we multiply the last row by 0.125 and add it to the first row. Afterwards, we multiply again the last row by 0.4375 and this time subtracts it from the second one.

$$\left(\begin{array}{ccc|c} 1 & 0 & -0.125 & 3.75 \\ 0 & 1 & 0.4375 & -0.625 \\ 0 & 0 & 1 & 170 \end{array}\right) \to \left(\begin{array}{ccc|c} 1 & 0 & 0 & 25 \\ 0 & 1 & 0.4375 & -0.625 \\ 0 & 0 & 1 & 170 \end{array}\right)$$

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & 25 \\ 0 & 1 & 0.4375 & -0.625 \\ 0 & 0 & 1 & 170 \end{array}\right) \to \left(\begin{array}{ccc|c} 1 & 0 & 0 & 25 \\ 0 & 1 & 0 & -75 \\ 0 & 0 & 1 & 170 \end{array}\right)$$

We reduced the augmented matrix to the reduced row echelon form. Let's transform back the problem to the system of linear equations.

$$\begin{cases} 1 \cdot x + 0 \cdot y + 0 \cdot z = x = 25 \\ 0 \cdot x + 1 \cdot y + 0 \cdot z = y = -75 \\ 0 \cdot x + 0 \cdot y + 1 \cdot z = z = 170 \end{cases}$$

We now do posses a tool (algorithm) for solving a system of linear equations.[9]

Challenge 3 *We discovered a solution to the system of linear equations by just following an algorithm and by using only* 3 *main operations. As an exercise, and by applying the newly discovered method of solving system of linear equations, can you solve this challenge? The system of equations is listed in fig.* 5.1 *on the next page.*

Definition 1 *Some but not all quadratic matrices have inverses, that is for* $A = (a_{i,j})$ *a matrix* A^{-1} *such that*

$$A \cdot A^{-1} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

*is called the **inverse** of A. If A has an inverse matrix, A is said to be **invertible**.*

[9]How to do that with *SageMath* is described below in section 7 on page 16.

$$\begin{cases} 115\,b + 111\,h + 108\,f = 2209 \\ 118\,b + 101\,h + 115\,f = 2214 \\ 111\,b + 114\,h + 116\,f = 2286 \end{cases}$$

$$\begin{cases} 97\,q + 100\,m + 100\,a = 1582 \\ 111\,q + 110\,m + 101\,a = 1748 \\ 116\,q + 111\,m + 101\,a = 1786 \end{cases}$$

$$\begin{cases} 97\,r + 99\,n + 104\,t = 910 \\ 108\,r + 101\,n + 116\,t = 1005 \\ 116\,r + 101\,n + 114\,t = 1019 \end{cases}$$

Figure 5.1: Puzzle Challenge 3

If A has such an inverse, it is unique and the outcome of the product $A \cdot A^{-1} = A^{-1} \cdot A$ doesn't depend on the order of the factors. Mind that matrix multiplication in general is not commutative.

Of course we can easily compute the inverse of a matrix in *SageMath*, if it exists:

```
sage: A=matrix([[0,2,0,0],[3,0,0,0],[0,0,5,0],[0,0,0,7]])
sage: A
[0 2 0 0]
[3 0 0 0]
[0 0 5 0]
[0 0 0 7]
sage: A.inverse()
[  0 1/3   0   0]
[1/2   0   0   0]
[  0   0 1/5   0]
[  0   0   0 1/7]
sage: B=matrix([[1,0],[0,0]])
sage: B.inverse()
#... lines of error info, ending with:
ZeroDivisionError: matrix must be nonsingular
```

Definition 2 *A* ***diagonal matrix*** *is a matrix $A = (a_{ij})$ where $a_{ij} = 0$ for all i, j with $i \neq j$. That is, all entries outside the diagonal are zero.*

Note that matrix multiplication, restricted to diagonal matrices, is commutative. Note

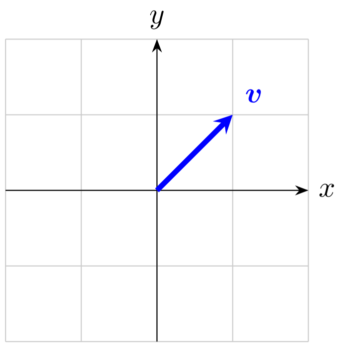


Figure 6.1: Example of a vector

also that transposition operates trivially on diagonal matrices: $A^T = A$ for every diagonal matrix A.

In order to continue our path to the definition of lattices and their properties, we need to introduce some basic definitions and notations about vectors, which – depending on the context – sometimes can be viewed as special matrices either with only one column or with only one row.

6 Vectors

A **scalar** quantity is a one dimensional measurement of a quantity, like temperature or mass. A **vector** has more than one number associated with it. A simple example is velocity. It has a magnitude, called speed, as well as a direction, like North or Southwest or 10 degrees west of North. You can have more than two numbers associated with a vector.[10] We often draw a vector as an arrow as shown in fig. 6.1 – here a vector v is drawn starting from the origin $(0, 0)$ and terminating on point $(1, 1)$. But how to write down the vector?

Definition 3 *A directed line from the point $P(x_1, x_2)$ to the point $Q(y_1, y_2)$ is a* ***vector*** *with the following components: $\overrightarrow{PQ} = \overrightarrow{OS} = (s_1, s_2) = (y_1 - x_1, y_2 - x_2)$.*

The initial point of the vector $\overrightarrow{OP} = (x_1, x_2)$ is at the origin $O = (0, 0)$ and the terminal point is $P = (x_1, x_2)$.[11]

Let's express the vectors $\overrightarrow{PQ}$ and $\overrightarrow{RQ}$ having the three points $P(0, 1)$, $Q(2, 2)$ and $R(1.5, 1.5)$ as shown in fig. 6.2 on the next page.

[10] This explanation is taken from van.physics.illinois.edu.

[11] This and some other useful definitions and notations are taken from the freely available book "Linear Algebra with Sage" from Sang-Gu Lee[6].

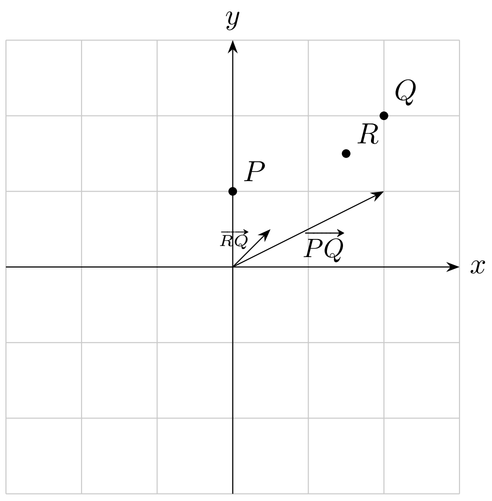


Figure 6.2: Finding vectors

We can easily do that by following the definition:

$$\overrightarrow{PQ} = (2-0, 2-1) = (2,1)$$

$$\overrightarrow{RQ} = (2-1.5, 2-1.5) = (0.5, 0.5)$$

Moreover, if we define the origin point as $O(0,0)$ and some random point $Z(x,y)$ we can easily define the vector $\overrightarrow{OZ} = (x-0, y-0) = (x,y)$. Using this observation we can easily calculate the desired vectors using *SageMath*.

```
sage: vOP = vector([0,1])
sage: vOQ = vector([2,2])
sage: vOR = vector([1.5,1.5])
sage: vPQ = vOQ - vOP
sage: vRQ = vOQ - vOR
sage: print(vPQ, vRQ)
(2,1) (0.5, 0.5)
```

Intuitively, we can easily check that results. $\overrightarrow{PQ} = (2,1)$ means that if we start moving from point P and move 2 times on the right and 1 time up, we are going to reach the point Q. Following the same interpretation, $\overrightarrow{RQ} = (0.5, 0.5)$ means that if we start moving from point R and move 0.5 on the right and 0.5 up, we are going to reach the point Q.

Definition 4 ***(Addition of vectors, multiplication of a scalar with a vector)***

For any two vectors $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, $y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$ *in* R^2 *and a scalar* k*, the sum of* $x + y$ *and the product* kx *are defined as follows:* $x + y = \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \end{bmatrix}$ *and* $kx = \begin{bmatrix} kx_1 \\ kx_2 \end{bmatrix}$.

Definition 5 *The **zero vector** is a vector where all its components are equal to* 0 *(its initial point is taken to be the origin).*

Definition 6 *An **ordered n-tuple** of (real) numbers* $(x_1, x_2, ..., x_n)$ *is called a* n*-dimensional vector and can be written as*

$$x = (x_1, x_2, ..., x_n) = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdots \\ x_n \end{bmatrix}$$

We call $x_1, x_2, ..., x_n$ *the **components** of* x*.*

We sum n-dimensional vectors and multiply n-dimensional vectors by some scalar the same way as we did with the 2-dimensional ones.

Definition 7 *For vectors* $v_1, v_2, ..., v_k$ *in* R^n *and scalars* $c_1, c_2, ..., c_k$*,*

$$x = c_1v_1 + c_2v_2 + ...c_kv_k$$

*is called a **linear combination** of vectors* $v_1, v_2, ..., v_k$*.*

Definition 8 *Given a vector* $x = (x_1, x_2, ..., x_n)$ *in* R^n

$$\|x\| = \sqrt{x_1^2 + x_2^2 + ... + x_n^2}$$

*is called the **norm (or length)** of* x*.*

You can easily calculate the norm of the vector by using *SageMath*:

```
sage: v = vector([3,6,2])
sage: v.norm()   # square root of (9 + 36  + 4)
7
```

Challenge 4 *As an exercise, find the hidden word in the puzzle challenge, given in fig. 6.3. Hint: 0xASCII.*

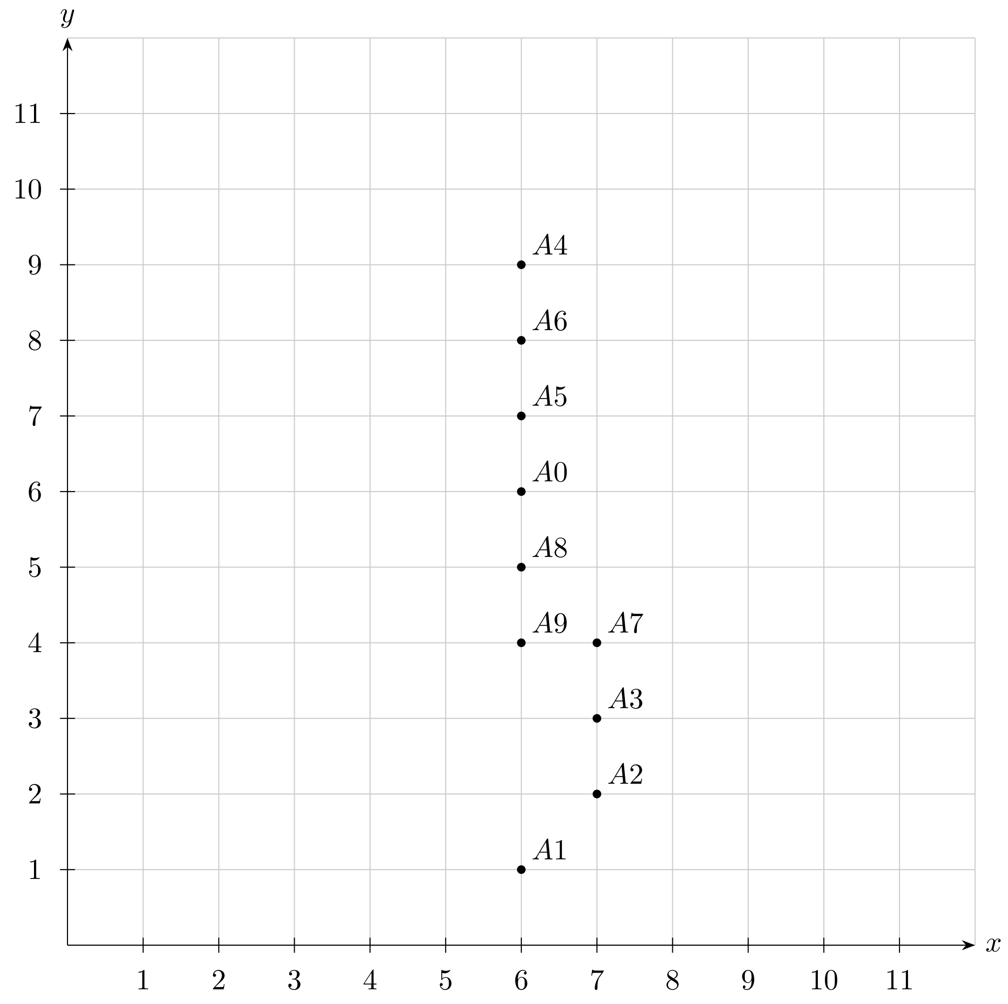


Figure 6.3: Puzzle Challenge 4

Definition 9 *For vectors $x = (x_1, x_2, ..., x_n)$, $y = (y_1, y_2, ..., y_n)$ in R^n,*

$$x_1y_1 + x_2y_2 + ... + x_ny_n$$

is called the ***dot product*** *or* ***scalar product*** *or* ***inner product***[12] *of x and y and is denoted by $x \cdot y$.*

We are not going to use the following 3 definitions 10, 11, and 12 in the sections to come,

[12]To be precise, the dot product is a subset of the inner product, but this difference isn't used here.

but they are handy consequences of the previous definitions.

Definition 10 *For nontrivial vectors $x = (x_1, x_2, ..., x_n), y = (y_1, y_2, ..., y_n)$ in R^n there exists θ with $0 \leq \theta \leq \pi$ or $0^\circ \leq \theta \leq 180^\circ$ and $\frac{x \cdot y}{||x|| \cdot ||y||} = \cos\theta$. Then θ is called the **angle** between x and y.*

Definition 11 *If $x \cdot y = 0$, then x is **orthogonal** to y. If x is a scalar multiple of y, then x is **parallel** to y.*

We can easily calculate the inner product of two vectors via *SageMath*.

```
sage: x = vector([5,4,1,3])
sage: y = vector([6,1,2,3])
sage: x*y
45
sage: x.inner_product(y)
45
```

We can either use the multiply operator or be more strict and use the second syntax. Indeed, $x \cdot y = 5 \cdot 6 + 4 \cdot 1 + 1 \cdot 2 + 3 \cdot 3 = 45$. Now, it's time to define the building blocks of one vector space.

Definition 12 *For an arbitrary, non-zero vector $v \in R^n$, $u = \frac{1}{||v||} \cdot v$ is a **unit vector**. In R^n, unit vectors of the form:*

$$e_1 = (1, 0, 0, ..., 0), e_2 = (0, 1, 0, ..., 0), ..., e_n = (0, 0, 0, ..., 1)$$

*are called **standard unit vectors** or coordinate vectors.*

The previous definition allows us to make the following observation: If $x = (x_1, x_2, ..., x_n)$ is an arbitrary vector of R^n, using standard unit vectors, we can express x as follows:

$$x = x_1e_1 + x_2e_2 + ... + x_ne_n$$

7 Equations – revisited

We introduced the concept of matrices and more precisely – the coefficient matrix and the augmented matrix (see section 5 on page 7). We examined the Gaussian elimination and how to solve a system of linear equations by using it.

So, let's solve the following system of linear equations using *SageMath*:

$$\begin{cases} 96x_1 + 11x_2 + 101x_3 = 634 \\ 97x_1 + 15x_2 + 99x_3 = 637 \\ 88x_1 + 22x_2 + 100x_3 = 654 \end{cases}$$

Let's first define the coefficient matrix A.

```
sage: A = matrix([[96,11,101],[97,15,99],[88,22,100]])
sage: A
[ 96  11 101]
[ 97  15  99]
[ 88  22 100]
```

Then, we define the right sides of the equations as a vector b and construct the augmented matrix.

```
sage: b = vector([634,637,654])
sage: b
(634, 637, 654)
sage: B = A.augment(b)
sage: B
[ 96  11 101 634]
[ 97  15  99 637]
[ 88  22 100 654]
```

Now, all we need to do is to directly calculate the reduced row echelon form.

```
sage: B.rref()
[1 0 0 1]
[0 1 0 3]
[0 0 1 5]
```

As a final solution, we have $x_1 = 1, x_2 = 3, x_3 = 5$.

We already defined operations for dealing with vectors (see definition 4 on page 13 and definition 9 on page 15). Let's apply the same operations when we are dealing with matrices.

Definition 13 *(**Addition**) Given two matrices $A = [a_{ij}]_{m\times n}$ and $B = [b_{ij}]_{m\times n}$ the sum of $A + B$ is defined by*

$$A + B = [a_{ij} + b_{ij}]_{m\times n}$$

Definition 14 *Given a matrix $A = [a_{ij}]_{m\times n}$ and a real number k, the* ***scalar multiple*** *kA is defined by*

$$kA = [ka_{ij}]_{m\times n}$$

Let's try some examples with *SageMath*:

```
sage: A = matrix([[9,7,0],[0,5,6],[1,3,3]])
sage: B = matrix([[8,5,2],[8,2,2],[0,0,1]])
sage: A
[9 7 0]
[0 5 6]
[1 3 3]
sage: B
[8 5 2]
[8 2 2]
[0 0 1]
sage: A+B
[17 12  2]
[ 8  7  8]
[ 1  3  4]
sage: A-B
[ 1  2 -2]
[-8  3  4]
[ 1  3  2]
sage: 2*A
[18 14  0]
[ 0 10 12]
[ 2  6  6]
sage: A.row(0)   # get first row of a matrix
(9, 7, 0)
sage: A.column(0)   # get first col of a matrix
(9, 0, 1)
```

Definition 15 *Given two matrices $A = [a_{ij}]_{m\times p}$ and $B = [b_{ij}]_{p\times n}$, we define the* ***product*** *AB of A and B, s.t. $AB = [c_{ij}]_{m\times n}$, where*

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + .. + a_{ip}b_{pj}$$

Remark: Please note that the number of columns of the first factor must be equal to the number of rows in the second factor.

```
sage: A*B
[128  59  32]
[ 40  10  16]
[ 32  11  11]
```

We can conclude with the following observations from the definition of a product of two matrices: The inner product (see definition 9 on page 15) of the i-th row vector of A and the j-th column vector of B is the (i, j) entry of AB. To demonstrate this observation we first need to introduce another simple definition:

Definition 16 *The* ***transpose*** *of a matrix is a new matrix whose rows are the columns of the original, i.e. if* $A = [a_{ij}]_{m\times n}$*, then* A^T*, the transpose of* A*, is* $A^T = [a_{ji}]_{n\times m}$.

```
sage: B.transpose()
[8 8 0]
[5 2 0]
[2 2 1]
sage: B.T
[8 8 0]
[5 2 0]
[2 2 1]
```

Using this handy *SageMath* method we can easily calculate the product of two matrices:

```
sage: for a in A:
....:     for b in B.transpose():
....:         print(a*b)
....:     print()
....:
128
59
32

40
10
16

32
11
11
```

or directly in *SageMath*[13]

```
sage: A*B
[128  59  32]
[ 40  10  16]
[ 32  11  11]
```

In order to introduce the concept of **lattices** we need some more definitions. Considering the next system of equations, let's try to express each of the variables x, y and z as an expression of the coefficients $a, b, c, d, e, f, g, h, i, r_1, r_2, r_3$.

$$\begin{aligned} ax + by + cz &= r_1 \\ dx + ey + fz &= r_2 \\ gx + hy + iz &= r_3 \end{aligned} \tag{1}$$

Let's multiply the first equation in system 1 with ei, the second equation with hc and the last equation with bf.

$$\begin{aligned} aeix + beiy + ceiz &= eir_1 \\ dhcx + ehcy + fhcz &= hcr_2 \\ gbfx + hbfy + ibfz &= bfr_3 \end{aligned} \tag{2}$$

Again, using system 1, we multiply the first equation with fh, the second equation with bi and the last equation with ce.

$$\begin{aligned} afhx + bfhy + cfhz &= fhr_1 \\ dbix + ebiy + fbiz &= bir_2 \\ gcex + hcey + icez &= cer_3 \end{aligned} \tag{3}$$

Now we derive a new equation by subtracting from the sum of all the equations in system 2 the sum of all the equations in system 3:

$$\begin{aligned} &(aei + dhc + gbf - afh - dbi - gce) \cdot x + \\ &\qquad + (bei + ehc + hbf - bfh - ebi - hce) \cdot y + \\ &\qquad + (cei + fhc + ibf - cfh - fbi - ice) \cdot z = \\ &\qquad\qquad = (ei - fh) \cdot r_1 + (hc - bi) \cdot r_2 + (bf - ce) \cdot r_3 \end{aligned}$$

[13]Please note the preference of the *SageMath* operators: `A*B.transpose() = A*(B.transpose())` and not `(A*B).transpose()`

Simplifying the equation by removing the equal expressions:

$$\begin{aligned}(aei + dhc + gbf - afh - dbi - gce) \cdot x = \\ = (ei - fh) \cdot r_1 + (hc - bi) \cdot r_2 + (bf - ce) \cdot r_3\end{aligned}$$

So, we are ready to express x:

$$x = \frac{(ei - fh) \cdot r_1 + (hc - bi) \cdot r_2 + (bf - ce) \cdot r_3}{aei + dhc + gbf - afh - dbi - gce} \tag{4}$$

Following the same procedure, and choosing carefully the coefficients to multiply the equations with, we can express y and z as well. But what if we have had system of 100 equations with 100 variables? Moreover, how to use a more elegant way of recovering the variables? It's time to introduce the definitions of **minors** and **determinants**.

Definition 17 *A* ***minor*** M_{ij} *of a square matrix A with size n, is the $(n-1) \times (n-1)$ matrix made by the rows and columns of A except the i'th row and the j'th column.*

Example 1 *Let's have a matrix* $A = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$. *Following the definitions of minors, we have* $M_{11} = \left(\begin{smallmatrix} e & f \\ h & i \end{smallmatrix}\right)$, $M_{12} = \left(\begin{smallmatrix} d & f \\ g & i \end{smallmatrix}\right)$ *or* $M_{22} = \left(\begin{smallmatrix} a & c \\ g & i \end{smallmatrix}\right)$.

Let's take the general case of a matrix A with size n.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,j} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,j} & \cdots & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,j} & \cdots & a_{3,n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{i,1} & a_{i,2} & a_{i,3} & \cdots & a_{i,j} & \cdots & a_{i,n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,j} & \cdots & a_{n,n} \end{pmatrix}$$

Then, the minor M_{ij} of A is equal to:

$$M_{ij} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,j-1} & a_{1,j+1} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,j-1} & a_{2,j+1} & \cdots & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,j-1} & a_{3,j+1} & \cdots & a_{3,n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \\ a_{i-1,1} & a_{i-1,2} & a_{i-1,3} & \cdots & a_{i-1,j-1} & a_{i-1,j+1} & \cdots & a_{i-1,n} \\ a_{i+1,1} & a_{i+1,2} & a_{i+1,3} & \cdots & a_{i+1,j-1} & a_{i+1,j+1} & \cdots & a_{i+1,n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,j-1} & a_{n,j+1} & \cdots & a_{n,n} \end{pmatrix}$$

Now, having the definitions of minors, we can finally define the **determinant** of a matrix.

Definition 18 *Let's have a square matrix A with real number elements and some fixed integers $r \in \{1, \dots, n\}$ and $c \in \{1, \dots, n\}$. Then, its **determinant**, $det(A) = \left|A\right|$ is a real number, which can be calculated either by column c or by row r:*

$$\left|A\right| = \sum_{i=1}^{n} (-1)^{i+c} a_{i,c} \left|M_{ic}\right| \qquad \textit{(expansion along column c)}$$
$$\left|A\right| = \sum_{i=1}^{n} (-1)^{r+i} a_{r,i} \left|M_{ri}\right| \qquad \textit{(expansion along row r)}$$

Let's calculate the determinants of a 2×2 matrix B and of a 3×3 matrix C by using minors respectively on row 1 and column 2.

Example 2 *Expansion along row 1:*

$$det(B) = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = (-1)^{1+1} \cdot a \cdot \left|d\right| + (-1)^{1+2} \cdot b \cdot \left|c\right| = ad - bc$$

Note that in the above computation, $|d|$ and $|c|$ do not denote the absolute value of the numbers d and c, but the determinant of the 1×1-Matrices consisting of d or c, the minors $d = M_{11}$ and $c = M_{12}$ of the Matrix B.

Example 3 *Expansion along column 2:*

$$\begin{aligned} det(C) &= \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} \\ &= (-1)^{1+2} \cdot b \cdot \begin{vmatrix} d & f \\ g & i \end{vmatrix} + (-1)^{2+2} \cdot e \cdot \begin{vmatrix} a & c \\ g & i \end{vmatrix} + (-1)^{3+2} \cdot h \cdot \begin{vmatrix} a & c \\ d & f \end{vmatrix} \\ &= -b(di - fg) + e(ai - cg) - h(af - cd) = \\ &= aei + dhc + gbf - afh - dbi - gce \end{aligned}$$

The determinant of this example is exactly equal to the denominator of the right side of equation 4 on page 21. What about the numerator? We can easily verify that the numerator is equal to the determinant of matrix $B_1 = \begin{vmatrix} r_1 & b & c \\ r_2 & e & f \\ r_3 & h & i \end{vmatrix}$. If we define the matrices $B_2 = \begin{vmatrix} a & r_1 & c \\ d & r_2 & f \\ g & r_3 & i \end{vmatrix}$ and $B_3 = \begin{vmatrix} a & b & r_1 \\ d & e & r_2 \\ g & h & r_3 \end{vmatrix}$, we can easily calculate the variables x, y, and z.

$$x = \frac{det(B_1)}{det(C)}, \quad y = \frac{det(B_2)}{det(C)}, \quad z = \frac{det(B_3)}{det(C)} \tag{5}$$

The same method can be applied when looking for some solution of a system of n equations with n variables. *SageMath* provides easy way to calculate a determinant, as well as sub-matrices of our choice.

```
sage: M = matrix([[1,2,3], [4,5,6], [7,8,9]])
sage: M
[1 2 3]
[4 5 6]
[7 8 9]
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

```
sage: M.determinant()
0
sage: det(M)
0
sage: M.det()
0
```

```
# consider row 1 and 3, and col 1 and 3 (row 1 has index 0]
sage: M.matrix_from_rows_and_columns([0,2],[0,2])
[1 3]
[7 9]
```

$$M_{22} = \begin{pmatrix} 1 & 3 \\ 7 & 9 \end{pmatrix}$$

The constructor `matrix_from_rows_and_columns` takes two lists as arguments. The first list defines which rows of a matrix A should be taken to construct the new matrix, while the second list defines the columns. For example, to construct the minor (see def. 17) M_{ij} of matrix A with size n, we call:

```
A.matrix_from_rows_and_columns([0,...,i-1,i+1,...,n-1],[0,...,j-1,j+1,...,n-1])
```

Now, with all this information we can automate the process of solving a large system of equations. For example, we can construct a matrix from all the coefficients in the equations. Then, we calculate each minor and it's corresponding determinant, which will help us recovering all the variables as shown in 5 on 23. Let's consider the following system of equations:

$$\begin{cases} x + 9y + 3z = 61 \\ 2x + 4y + 8z = 94 \\ 5x + 7y + 6z = 128 \end{cases}$$

We can transfer the same system of equations into a product of matrices:

$$\begin{pmatrix} 1 & 9 & 3 \\ 2 & 4 & 8 \\ 5 & 7 & 6 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 61 \\ 94 \\ 128 \end{pmatrix}$$

Now, for example, we can use determinants to calculate the unknowns. On the other hand, we can further automate this process by using *SageMath*:

```
sage: M = matrix([[1,9,3], [2,4,8], [5,7,6]])
sage: M
```

$$\begin{pmatrix} 1 & 9 & 3 \\ 2 & 4 & 8 \\ 5 & 7 & 6 \end{pmatrix}$$

```
sage: r = matrix([[61],[94],[128]])
sage: r
```

$$\begin{pmatrix} 61 \\ 94 \\ 128 \end{pmatrix}$$

```
sage: M.solve_right(r)
```

$$\begin{pmatrix} 13 \\ 3 \\ 7 \end{pmatrix}$$

Which yields the final solutions $x = 13$, $y = 3$ and $z = 7$.

Challenge 5 *Alice and Bob established an interesting (but insecure) encryption scheme. Alice creates n equations with n variables and sends them to Bob over an insecure channel using two packets. The first packet consists of all the coefficients used in the equations* *in form of a matrix without any changes. However, the second packet consists of all the right sides of the equations in scrambled order. Their shared secret key consists of the original indexes of the scrambled right sides of the equations.*

Having the secret key, Bob can unscramble the right sides of the equations and recover the unknown variables. Then, he multiplies all the recovered variables and the final number is the decrypted message. They were using `leet` *language to create or read the final number. For example, the word* `sage` *in leet language is* `5463`*.*

Eve captured the following two packets P_1 and P_2:

$$P_1 = \begin{pmatrix} 33 & 79 & 29 & 41 & 47 \\ 79 & 27 & 39 & 79 & 44 \\ 90 & 83 & 58 & 1 & 90 \\ 38 & 32 & 13 & 15 & 96 \\ 72 & 82 & 88 & 83 & 23 \end{pmatrix}; \qquad P_2 = [\,73\,300,\ 167\,887,\ 243\,754,\ 254\,984,\ 458\,756\,]$$

Can you recover the original message?

8 Vector Spaces

We need to introduce another important building block of linear algebra – *vector spaces*. Let's first define what a vector space is.

Definition 19 *A* ***vector space*** *over the real numbers* $\mathbb{R}$ *consists of a set* V *and two operators* $\oplus$ *and* $\odot$*, subject to the following conditions/properties/axioms:*[14]

1. *closure under addition:* $\forall \vec{x}\, \forall \vec{y} \in V : \vec{x} \oplus \vec{y} \in V$
2. *commutativity of addition:* $\forall \vec{x}\, \forall \vec{y} \in V : \vec{x} \oplus \vec{y} = \vec{y} \oplus \vec{x}$
3. *associativity of addition:* $\forall \vec{x}\, \forall \vec{y} \forall \vec{z} \in V : (\vec{x} \oplus \vec{y}) \oplus \vec{z} = \vec{x} \oplus (\vec{y} \oplus \vec{z})$
4. *neutral element under addition:* $\exists\, \vec{o} \in V : (\forall \vec{x} \in V : \vec{x} \oplus \vec{o} = \vec{x})$ *We call* $\vec{o}$ *the* zero vector*.*
5. *additive inverse:* $\forall \vec{x} \in V\ \exists \vec{y} \in V : \vec{x} \oplus \vec{y} = \vec{o}$ *(We call* $\vec{y}$ *the* additive inverse or inverse element of addition *of* $\vec{x}$*, and vice versa.)*
6. *closure under scalar multiplication:* $\forall r \in \mathbb{R}, \forall \vec{x} \in V : r \odot \vec{x} \in V$
7. *distributivity:* $\forall r, s \in \mathbb{R}, \forall \vec{x} \in V : (r + s) \odot \vec{x} = r \odot \vec{x} \oplus s \odot \vec{x}$
8. *distributivity:* $\forall r \in \mathbb{R}, \forall \vec{x}, \vec{y} \in V : r \odot (\vec{x} \oplus \vec{y}) = r \odot \vec{x} \oplus r \odot \vec{y}$
9. *associativity:* $\forall r, s \in \mathbb{R}, \forall \vec{x} \in V : (rs) \odot \vec{x} = r \odot\ (s \odot \vec{x})$
10. *neutral or identity element of scalar multiplication:* $\forall \vec{x} \in V : 1 \odot \vec{x} = \vec{x}$

Usually, one does not use the symbol $\odot$, but just the multiplication dot or even no dot. But sometimes, like in this definition, we want to point out that the scalar multiplication is a mapping from $\mathbb{R} \times V$ to V while the "regular" multiplication is something else, i.e. a mapping from $\mathbb{R} \times \mathbb{R}$ to $\mathbb{R}$. Also note that there is some danger of confusion when we use the same multiplication symbol (or no symbol at all) for scalar multiplication as well as for the scalar product (vector product, inner product, see page 15). This can cause some trouble when looking at more complicated formulas containing both types of products.

We can further define vector spaces over larger sets, for example the set of complex numbers. It is also possible to define structures like above over smaller sets, for example the set of integer numbers. In this case – when the set of scalars is a so called *ring*[15] and not a so called *field* – these algebraic structures are not called vector spaces, but *modules*. Our vectors, as well as our choice of the two operators $\oplus$ and $\odot$ play an important role whether our space is a vector space or not.

[14] See the good Wikipedia article https://en.wikipedia.org/wiki/Vector_space about that topic.

[15] We will not go into detail here and define what a field or a ring is. Basically, a field is something like $\mathbb{R}$ and a ring is something like $\mathbb{Z}$. The main difference is that a field has multiplicative inverses for all elements except zero and a ring doesn't.

Example 4 *Let's define the set M of all 2×2 matrices with entries of real numbers. Furthermore, we choose the operator $\oplus$ as a regular additive operator on matrices, i.e.,*

$$\begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \oplus \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix} = \begin{pmatrix} a_1+b_1 & a_2+b_2 \\ a_3+b_3 & a_4+b_4 \end{pmatrix}$$

We choose the operation $\odot$ to be the already known scalar multiplication of matrices, i.e.

$$r \odot \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} = \begin{pmatrix} ra_1 & ra_2 \\ ra_3 & ra_4 \end{pmatrix}$$

We can easily check that all the conditions apply and this is indeed a vector space, in which the zero vector is $\left(\begin{smallmatrix} 0 & 0 \\ 0 & 0 \end{smallmatrix}\right)$.

Example 5 *The set P of polynomials with real coefficients is a vector space with the operator $\oplus$ defined to be the regular additive operator on polynomials and the operator $\odot$ defined via $r \odot (\sum a_i x^i) := \sum r a_i x^i$. For example, if $a_i, b_i, r \in \mathbb{R}$, then:*

$$\begin{aligned} (a_0 + a_1 x + ... + a_n x^n) \quad &\oplus \quad (b_0 + b_1 x + ... + b_n x^n) = \\ &= \quad (a_0 + b_0) + (a_1 + b_1)x + ... + (a_n + b_n)x^n \\ r \odot (a_0 + a_1 x + ... + a_n x^n) \quad &= \quad (ra_0) + (ra_1)x + ... + (ra_n)x^n \end{aligned}$$

Definition 20 *For any vector space V over a given field with operations $\oplus$ and $\odot$, a* ***subspace*** *U is a subset of V that is itself a vector space over the same field under the inherited operations $\oplus$ and $\odot$. This means that U is closed under addition $\oplus$ and scalar multiplication $\odot$.*

Example 6 *A* ***subspace*** *of $\mathbb{R}^2$ is the set of the zero vector $\{(0,0)\}$. We call such subspaces* ***trivial subspaces****.*

Example 7 *Let's define the vector space of cubic polynomials*

$$C_{\text{u}} = \{a + bx + cx^2 + dx^3 \mid a, b, c, d \in \mathbb{R}\}$$

and the vector space of linear polynomials $L_{\text{i}} = \{e + fx | e, f \in \mathbb{R}\}$. Then, L_{i} is a subspace of C_{u}.

Definition 21 *The* **span** *(or* **linear closure***) of a nonempty subset S of a vector space V is the set of all linear combinations of vectors from S.*

In short, we can write down the span of a subset S of a vector space V in the following way:

$$span(S) = \{c_1\vec{v_1} \oplus ... \oplus c_n\vec{v_n} \mid c_i \in \mathbb{R},\, \vec{v_i} \in S\}$$

Note that S itself does not have to be a subspace, but $span(S)$ always is a subspace of V. If $span(S) = U$, we say that S generates U.

Example 8 *For any nonzero vector $\vec{x} \in \mathbb{R}^3$, the span of $\vec{x}$ is a line through the origin* $(0, 0, 0)$.

Example 9 *Let's define $S = \left\{\left(\begin{smallmatrix}2\\2\end{smallmatrix}\right), \left(\begin{smallmatrix}2\\-2\end{smallmatrix}\right)\right\}$. We will show that $span(S) = \mathbb{R}^2$.*

If this is the case, then each vector $\left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathbb{R}^2$ can be represented as a linear combination of vectors in S. Therefore $\exists\, r_1, r_2 \in \mathbb{R}$:

$$r_1 \cdot \begin{pmatrix}2\\2\end{pmatrix} + r_2 \cdot \begin{pmatrix}2\\-2\end{pmatrix} = \begin{pmatrix}x\\y\end{pmatrix}$$

So, we can express every possible vector $\left(\begin{smallmatrix}x\\y\end{smallmatrix}\right) \in \mathbb{R}^2$ by choosing r_1 and r_2 such that $r_1 = \frac{x+y}{4}$ and $r_2 = \frac{x-y}{4}$. For example, if $x = 9$ and $y = 1$, we have $r_1 = \frac{5}{2}$ and $r_2 = 2$.

$$\frac{5}{2} \cdot \begin{pmatrix}2\\2\end{pmatrix} + 2 \cdot \begin{pmatrix}2\\-2\end{pmatrix} = \begin{pmatrix}5\\5\end{pmatrix} + \begin{pmatrix}4\\-4\end{pmatrix} = \begin{pmatrix}9\\1\end{pmatrix}$$

The previous example is one possibility of spanning $\mathbb{R}^2$. Can the set $\mathbb{R}^2$ be spanned by 3 or more vectors? Sure, we can just duplicate one of the elements in the previous example, namely we can take the set $\left\{\left(\begin{smallmatrix}2\\2\end{smallmatrix}\right), \left(\begin{smallmatrix}2\\-2\end{smallmatrix}\right), \left(\begin{smallmatrix}1\\-1\end{smallmatrix}\right)\right\}$. But can $\mathbb{R}^2$ be spanned by using only one vector?

Definition 22 *A subset of a vector space is said to be* ***linearly independent*** *if none of its elements is a linear combination of the others. Otherwise it is called* ***linearly dependent****.*

Definition 23 *A* ***basis*** *for a vector space is a set B of vectors that is linearly independent and spans the space. If $|B| = n$ for some $n \in \mathbb{N}$ we define the* ***dimension*** *of B to be $dim(B) := n$. If $|B| = \infty$, the concept of dimension is also well defined, but the theory is somewhat more complicated because of different "types" of infinity in mathematics. We will not go into detail here.*

Example 10 *We already showed that the set $\left\{\left(\begin{smallmatrix}2\\2\end{smallmatrix}\right), \left(\begin{smallmatrix}2\\-2\end{smallmatrix}\right)\right\}$ is a basis of $\mathbb{R}^2$. Another one is $\left\{\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right), \left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)\right\}$.*

Example 11 *We can easily construct the basis ξ_n of $\mathbb{R}^n$ for any n:*

$$\xi_n = \left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \dots, \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix} \right\} =: \{e_1, \dots, e_n\}$$

*We say that this is the **standard** or **canonical** basis of $\mathbb{R}^n$.*

Every invertible Matrix $B = (b_{ij})$ induces a mapping from this standard basis to another basis $\{b_1, \dots, b_n\}$ with $b_i = \begin{pmatrix} b_{1i} \\ \vdots \\ b_{ni} \end{pmatrix}$, that is, every column of the matrix B can be seen as the the image of one of the canonical basis vectors. Using matrix multiplication, we have $B \cdot e_i = b_i$.

One special class of mappings between bases are the permutations. If we just change the order of the canonical basis vectors e_i, this is called a **permutation** and the corresponding matrix has only entries 0 or 1, exactly one 1 per row and line. For example mapping e_1 to e_3, e_3 to e_2 and e_2 to e_1 is done by

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}.$$

Such a matrix is called a permutation matrix. In algebra, permutations are sometimes written in parentheses like $(1, 3, 2)$ meaning to *map the first element to the third, the third element to the second and the second element to the first.* This notation is also used by sage. But be careful! The tuple $(1, 3, 2)$ in this context – sage notation `[1,3,2]` – can also mean *map the first element to the first element, the second element to the third element and the third element to the second.*

```
sage: per=Permutation((1,3,2))
sage: per
[3, 1, 2]
sage: matrix(per)
[0 1 0]
[0 0 1]
[1 0 0]
sage: grelt=PermutationGroupElement([1,3,2])
sage: grelt.matrix()
[1 0 0]
[0 0 1]
[0 1 0]
```

One interesting thing about those permutation matrices is that their inverse is identical with their transposed matrix:

```
sage: matrix(per).inverse()==matrix(per).T
True
```

Challenge 6 *Inspired by the concept of a basis of a vector space, Alice and Bob invented yet another cryptosystem. This time they use an encoding Ξ, which first encodes each letter to a predefined number. This number is equal to the index of the corresponding letter in the English alphabet. For instance, the word **Bob** is encoded in a number array* `[ 2, 15, 2 ]`.

Alice and Bob carefully chose a set of private keys K, they share as common pre-knowledge. Depending on the length of the message that should be sent, a distinct key is used. Let's define the key $k_m \in K$ as the key used for encrypting messages with length m.

Each key k_m is a $m \times m$ matrix, generated by the following rules:

- *Each element of the matrix is either 0 or a prime number between 100 and 999.*
- *The row vectors of the matrix are linearly independent.*
- *There are exactly m numbers that are different from 0.*

Now, the encryption E of a message M with length m is a straightforward procedure. For example, let's say that Alice encrypts this message M and sends it to Bob.

1. *Alice encodes M, so she has $\Xi(M)$.*
2. *Alice encrypts the encoded message with the corresponding key k_m, i.e.*

$$E(\Xi(M), k_m) = k_m * \Xi(M).$$

3. *The message $E(\Xi(M), k_m)$ is send via the insecure channel.*

Then, the decryption D of a message $E(\Xi(M), k_m)$ is done by Bob following those simple steps:

1. *The length of the encrypted message uniquely defines the key k_m Bob should use.*
2. *Bob constructs the decryption key d_m, by substituting each greater than zero element in the secret key k_m with it's reciprocal value.*[16]
3. *Then, Bob applies the decryption, i.e. $D(E(\Xi(M), k_m)) = E(\Xi(M), k_m) * d_m$.*
4. *Bob decodes back the decrypted message to recover the original one.*

Can you verify the correctness of this encryption scheme? Why or why not this decryption works? Can you recover the following encrypted word:

(6852, 3475, 17 540, 3076, 12 217, 6383, 745, 1347, 661, 6088, 15 354, 2384, 2097, 11 415, 3143)

[16]This so constructed matrix is not the inverse of k_m! The matrix k_m can be written as a product $k_m = P \cdot D$ with a diagonal matrix D and a permutation matrix P. Then $k_m^{-1} = (P \cdot D)^{-1} = D^{-1} \cdot P^{-1} = D^{-1} \cdot P^T$. Now when we construct the matrix d_m by substituting in k_m every nonzero element by its reciprocal value, we have $d_m = P \cdot D^{-1}$. If we look at the transposed matrix d_m^T, we have $d_m^T = (P \cdot D^{-1})^T = (D^{-1})^T \cdot P^T = D^{-1} \cdot P^{-1} = (P \cdot D)^{-1} = k_m^{-1}$. Instead of letting the matrix d_m^T operate from the left on a column vector, we can let the matrix d_m operate from the right on a row vector, since in general for matrices A and column vectors v and b there is an equivalence $Av = b \Leftrightarrow (Av)^T = b^T \Leftrightarrow v^T A^T = b^T$. This corresponds to the notation from above, where in the encryption process, $E(\Xi(M), k_m) = k_m * \Xi(M)$ means $k_m \cdot \Xi(M)$ and $\Xi(M)$ is treated as a column vector while in the decryption process, d_m is written on the right of $E(\Xi(M), k_m)$, so in this case $E(\Xi(M), k_m)$ is treated as a row vector and $E(\Xi(M), k_m) * d_m$ means $E(\Xi(M), k_m) \cdot d_m$.

9 Lattices

Now, we have all the building blocks in order to introduce the concept of lattices.

Definition 24 *Let $v_1, ..., v_n \in \mathbb{Z}^m$, $m \geq n$ be linearly independent vectors. An **integer lattice** L spanned by $\{v_1, ..., v_n\}$ is the set of all integer linear combinations of $v_1, ..., v_n$, such that:*

$$L = \left\{ v \in \mathbb{Z}^m \mid v = \sum_{i=1}^{n} a_i v_i,\ a_i \in \mathbb{Z} \right\} \tag{6}$$

Remark: **Linear** combination means that all a_i are integers. **Integer** lattice means that all v_{ij} (components of the vectors v_i) are integers, and so all dots in the infinite graphic have integer coordinates.

SageMath uses the notion of an **integral lattice**, which is a somewhat more complicated concept. Roughly speaking, in this case the components of the vectors $v_1, \ldots, v_n$ are not restricted to $\mathbb{Z}$, but can be arbitrary real numbers, while the “allowed” linear combinations still have to be integer linear combinations, that is, the coefficients a_i in $\sum_{i=1}^{n} a_i v_i$ have to be integers. Every integer lattice is an integral lattice.

The set of vectors $B = \{v_1, ..., v_n\}$ is called a **basis of the lattice** L. We also say that $L = L(B)$ is spanned by the vectors of the basis B.

We define the **dimension** of L as $dim(L) := n$.

In the case where $n = m$ we can canonically construct a quadratic matrix from the vectors of a lattice basis by writing them down row by row (or column by column). If we denote this matrix by M, we can compute the product $M \cdot M^T$, which is sometimes called the **Gram matrix of the lattice**. If this Gram matrix has only integer entries, the lattice is integral. Note that for going into detail here, we would have to introduce some more math, especially the theory of quadratic forms, symmetric bilinear forms etc., which are a kind of generalizition of the vector product introduced on page 15.

The example in figure 9.1a presents a 2-dimensional lattice with

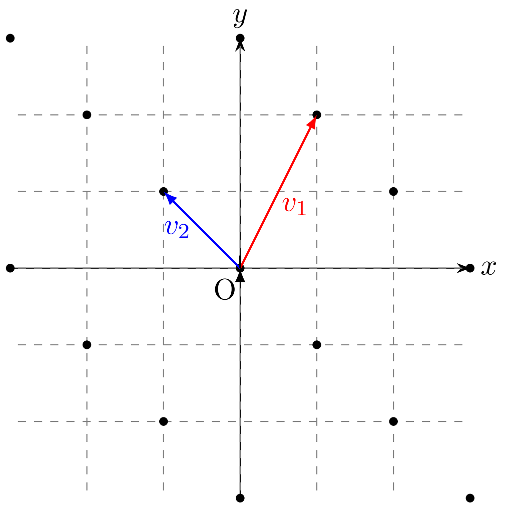
$$B_a = \{v_1, v_2\} = \{(1, 2), (-1, 1)\},$$

while the example in figure 9.1b presents a 2-dimensional lattice with

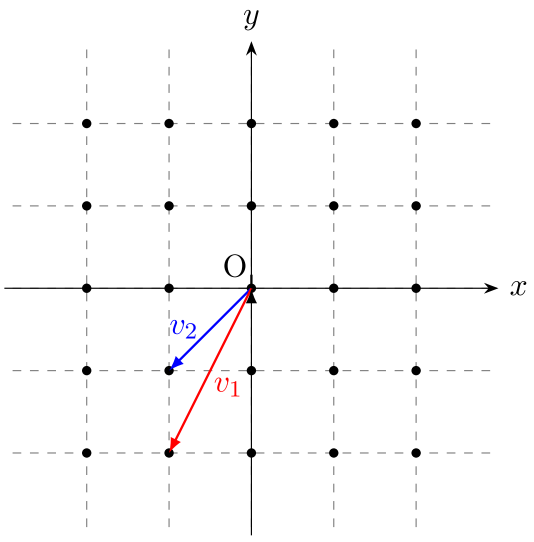
$$B_b = \{v_1, v_2\} = \{(-1, -2), (-1, -1)\}.$$[17]

We can informally generalize this definition by saying that the lattice L is formed by collecting all the points that are linear combinations of the defined basis B. In fact, the integer lattice is just the span of a regular integer matrix.

[17]Later on we show by comparing two spans of different lattices how the vectors of one basis can be converted to another one.



(a) A 2-dimensional lattice with basis B_a



(b) A 2-dimensional lattice with basis B_b

Figure 9.1: Example of lattices with different basis

Let's see how we can construct a lattice using *SageMath*:

```
sage: M = matrix(ZZ, [[1,2], [-1,1]])
sage: M
```

$$\begin{pmatrix} 1 & 2 \\ -1 & 1 \end{pmatrix}$$

Now, we can easily check if a given point (z_1, z_2) belongs to the lattice. If $(z_1, z_2) \in L$, then it belongs to the span of L.

```
sage: vector([1,1]) in span(M)
False

sage: vector([1,2]) in span(M)  # true because [1,2]=1*x+0*y
True

sage: vector([-1,2]) in span(M)
False

sage: vector([-101,5]) in span(M)
True
```

What if we want to see the exact linear combination that generates the point (z_1, z_2)? This means we want to know how this point z is uniquely built by the given basis vectors `x=[1,2]` and `y=[-1,1]`.

```
sage: M.solve_left(vector([-101,5]))
(-32, 69)
```

And indeed:

```
sage: -32*M[0] + 69*M[1]
(-101, 5)
```

We can define the same lattice using different bases. For example, let's introduce a lattice M2 with the following basis:

```
sage: M2 = matrix(ZZ, [[1,2], [0,3]])
sage: M2
```

$$\begin{pmatrix} 1 & 2 \\ 0 & 3 \end{pmatrix}$$

Now, by using the **span** function we can easily compare the identity of two objects defined by different bases.

```
sage: span(M) == span(M2)
True
```

Screenshots from CT2 showing how vectors span 2-dim bases can be found in section 15.2 on page 81.

9.1 Merkle-Hellman knapsack cryptosystem

Now, let's create another definition, before we take a look into the Merkle-Hellman knapsack cryptosystem.

Definition 25 *Any set of different nonzero natural numbers is called a **knapsack**. Furthermore, if this set can be arranged in an increasing list, in such a way that every number is greater than the sum of all the previous numbers, we will call this list a **superincreasing** knapsack.*[18]

[18]See `https://en.wikipedia.org/wiki/Superincreasing_sequence`

Challenge 7 *Inspired by the definition of the superincreasing knapsack, Alice and Bob constructed another insecure own cryptosystem. Can you find the hidden word in this intercepted message shown in the following number sequence? Hint: Is the knapsack superincreasing? Why or why not? Each number is keeping a secret bit. (A copy-and-paste-ready version of these numbers is also available, see page 75.)*

0, 0, 1, 2, 3, 6, 12, 25, 49, 98, 197, 394, 787, 1574, 3148, 6296,
12 593, 25 185, 50 371, 100 742, 201 484, 402 967, 805 935, 1 611 870,
3 223 740, 6 447 479, 12 894 959, 25 789 918, 51 579 835, 103 159 670,
206 319 340, 412 638 681, 825 277 361, 1 650 554 722, 3 301 109 445,
6 602 218 890, 13 204 437 779, 26 408 875 558, 52 817 751 117, 105 635 502 233,
211 271 004 467, 422 542 008 933, 845 084 017 867, 1 690 168 035 734,
3 380 336 071 467, 6 760 672 142 934, 13 521 344 285 869, 27 042 688 571 737,
54 085 377 143 475

The **Merkle-Hellman knapsack cryptosystem** is another **asymmetric** cryptosystem, theoretically interesting because it basically allows sending sensitive information over an insecure channel.

It consists of two knapsack keys:

- Public key – it's used **only for encryption**. It's called **hard** knapsack.
- Private key – it's used **only for decryption**. It consists of a **superincreasing** knapsack, a multiplier and a modulus. Multiplier and modulus can be used to convert the superincreasing knapsack into the hard knapsack.

The key generation algorithm performs the following steps:

- We first create a superincreasing knapsack $W = [w_1, w_2, \cdots, w_n]$.
- We choose an integer number q which is greater than the sum of all elements in W, i.e.

  $$q > \sum_{i=1}^{n} w_i$$

  We define q as the **modulus** of our cryptosystem.
- We pick a number r, s.t.

  $$r \in [1, q)$$
  $$(r, q) = 1$$

  where (r, q) is notation for the greatest common divisor (gcd) of r and q. We define r as the **multiplier** of our cryptosystem.

- The **private key** of the cryptosystem consists of the tuple (W, r, q).
- We generate the sequence $H = [h_1, h_2, \cdots, h_n]$, s.t. $h_j = w_j * r \mod q$, for $1 \leq j \leq n$. We define H as the **public key** of the cryptosystem.

If we want to encrypt a message m, we first take its bit representation $B_m = \overline{m_1 m_2 ... m_n}$, where m_i denotes the i–th bit, i.e. $m_i \in \{0, 1\}$. In order to ensure correctness of the algorithm, our superincreasing knapsack K should have at least n elements. Let's define it as $W = [w_1, w_2, ..., w_n, ...]$. Following the keys generation procedure, we generate its corresponding public key H, i.e. $H = [h_1, h_2, ..., h_n, ...]$ with some appropriate q and r. Then, the encryption c of m is the sum

$$c = \sum_{i=1}^{n} m_i h_i$$

If we want to decrypt the message c, we first calculate $c' = c * r^{-1} \mod q$, where r^{-1} is the modular inverse of r in q. Then, we start a procedure of decomposing c' by selecting the largest elements in W, which are less or equal to the remaining value which is currently being decomposed. Finally, we recover $m = \overline{m_1 m_2 ... m_n}$ by substituting m_j with 1 if the element w_j was chosen in the previous step. Otherwise m_j is equal to 0.

We intentionally only describe the pure algorithm as a cryptographic scheme (because of its weakness against lattice attacks) and do not discuss practical implementation issues like ensuring that the length of B_m is shorter or equal to the length of H or how and when padding has to be applied.

Example 12 *Let's say that Alice wants to encrypt and send the message* ***crypto*** *to Bob by using the Merkle-Hellman knapsack cryptosystem. Throughout this example all letters are to be handled independently. So n is always 8, as each letter has a binary representation of 8 bits.*

First, Bob needs to generate his private and public key. Bob initiates the process of creation of the private key by first generating a superincreasing knapsack W:

$$W = [11, 28, 97, 274, 865, 2567, 7776, 23253]$$

Then, Bob generates the corresponding modulus q and multiplier r:

$$q = 48433 > \sum_{i=1}^{n} w_i = 34871$$

$$r = 2333 < q$$

$$(2333, 48433) = 1$$

So, Bob composes the private key $P_r = (W, r, q)$:

$$P_r = ([11, 28, 97, 274, 865, 2567, 7776, 23253], 2333, 48433)$$

The final step for Bob is to generate the hard knapsack H and the public key $P_u = (H)$ and deliver it to Alice:

$$H = [25663, 16891, 32569, 9613, 32292, 31552, 27466, 4289]$$

$$P_u = ([25663, 16891, 32569, 9613, 32292, 31552, 27466, 4289])$$

*Before encrypting the message $M =$ **crypto**, Alice divides the message into single letters and substitutes each letter with its own bit representation, i.e:*

$$\begin{aligned} c &= 01100011 \\ r &= 01110010 \\ y &= 01111001 \\ p &= 01110000 \\ t &= 01110100 \\ o &= 01101111 \end{aligned}$$

*Now, Alice calculates for the bit representation of each letter the corresponding encrypted number by using the public key H. So the algorithm has to be applied 6 times. Finally, the list of encrypted numbers C of the word **crypto** is:*

$$C = [81215, 86539, 95654, 59073, 90625, 145059]$$

When Bob receives C he first calculates C' by using r and q from the P_r.

$$C' = [31154, 8175, 24517, 399, 2966, 34586]$$

Then, by using W from P_r, he represents each element in C' as a sum of elements in W, following the above-mentioned algorithm. For example, let's decompose 31154*. With* ✓ *we will denote those elements in W which participate in the decomposition of* 31154 *and with* ✗ *those which don't. The sign * will denote the unknowns. We have:*

$$\begin{array}{l} [11, 28, 97, 274, 865, 2567, 7776, 23253] \\ [*,\ *,\ *,\ *,\ *,\ \ *,\ \ \ *,\ \ \ \ *\ ], 31154 \end{array} \tag{7}$$

The largest number in W smaller than 31154 *is* 23253*. We mark it as an element used in the decomposition of* 31154 *and we continue with the decomposition of the remaining* *value* 7901 = 31154 − 23253*:*

$$[11, 28, 97, 274, 865, 2567, 7776, 23253]$$
$$[*,\ *,\ *,\ *,\ *,\ *,\ *,\ ✓\ ], 7901$$

The largest element smaller than 7901 *is* 7776*. We proceed with this algorithm until reaching* 0*.*

$$[11, 28, 97, 274, 865, 2567, 7776, 23253]$$
$$[*,\ *,\ *,\ *,\ *,\ *,\ ✓,\ ✓\ ], 125$$
$$[11, 28, 97, 274, 865, 2567, 7776, 23253]$$
$$[*,\ *,\ *,\ *,\ *,\ ✗,\ ✓,\ ✓\ ], 125$$
$$[11, 28, 97, 274, 865, 2567, 7776, 23253]$$
$$[*,\ *,\ *,\ *,\ ✗,\ ✗,\ ✓,\ ✓\ ], 125$$
$$[11, 28, 97, 274, 865, 2567, 7776, 23253]$$
$$[*,\ *,\ *,\ ✗,\ ✗,\ ✗,\ ✓,\ ✓\ ], 125$$
$$[11, 28, 97, 274, 865, 2567, 7776, 23253]$$
$$[*,\ *,\ ✓,\ ✗,\ ✗,\ ✗,\ ✓,\ ✓\ ], 28$$
$$[11, 28, 97, 274, 865, 2567, 7776, 23253]$$
$$[*,\ ✓,\ ✓,\ ✗,\ ✗,\ ✗,\ ✓,\ ✓\ ], 0$$

So, at the end 31154 *is decomposed to* 01100011*, which is the bit representation of the letter* ***c****. By applying the same decrypting algorithm to all the elements of C, Bob finally recovers the encrypted message* ***crypto****.*

Challenge 8 *Encrypting long messages by repeatedly using small length hard knapsack yield some risks. In the next puzzle, you have to recover the encrypted message Alice sent to Bob. The private and public key are different from the one generated in the previous example. However, you know that the length of H is the same as before: $n = 8$. Can* *you recover the message even without knowing the public key?*

$$\begin{gathered}
333644, 560458, 138874, 389938, 472518, 394128, 138874, 472518, 560458,\\
138874, 465914, 384730, 550286, 138874, 462498, 472518, 638226, 560458,\\
138874, 634810, 389938, 138874, 628828, 472518, 465914, 384730, 550286,\\
628828, 472518, 465914, 551060, 478500, 560458, 138874, 394128, 550286,\\
389938, 550286, 394128, 138874, 465914, 634810, 138874, 394128, 550286,\\
472518, 462498, 551060, 465914, 633018, 295184, 138874, 465914, 384730,\\
550286, 633018, 138874, 472518, 394128, 550286, 138874, 468480, 634810,\\
465914, 138874, 478500, 550286, 394128, 465914, 472518, 551060, 468480,\\
295184, 138874, 472518, 468480, 383956, 138874, 472518, 560458, 138874,\\
389938, 472518, 394128, 138874, 472518, 560458, 138874, 465914, 384730,\\
550286, 633018, 138874, 472518, 394128, 550286, 138874, 478500, 550286,\\
394128, 465914, 472518, 551060, 468480, 295184, 138874, 465914, 384730,\\
550286, 633018, 138874, 383956, 634810, 138874, 468480, 634810, 465914,\\
138874, 394128, 550286, 389938, 550286, 394128, 138874, 465914, 634810,\\
138874, 394128, 550286, 472518, 462498, 551060, 465914, 633018, 301166
\end{gathered}$$

A screenshot with a visualization of the Merkle-Hellman knapsack cryptosystem in JCT can be found in fig. 15.13 on page 88.

Screenshots from CT2 about a ready-to-run lattice-based attack against the Merkle-Hellman knapsack cryptosystem can be found in section 15.2 on page 81.

9.2 Lattice-based cryptanalysis

Encrypting a message using hard knapsack with length at least the length of the message is far more secure, but still vulnerable. We will demonstrate this vulnerability by using a specially designed lattice.

Having a public key with hard knapsack H and an encrypted message C, we can represent each element of H as a vector in $|H|$ dimensional lattice. We need $|H|$ dimensions in order to guarantee they form a basis of the lattice L. In order to guarantee they are linearly independent, we just augment the transpose H to the identity matrix with dimension $|H| - 1$.

As example, let's take H with length 8:

$$H = [h_1, h_2, \cdots, h_8]$$

Then, the constructed lattice will have the form:

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & h_2 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & h_3 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & h_4 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & h_5 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & h_6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & h_7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & h_8 \end{pmatrix}$$

All the rows are linearly independent. Furthermore, we add another row in the lattice by plugging in the encrypted number c as its last element.

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & h_2 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & h_3 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & h_4 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & h_5 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & h_6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & h_7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & h_8 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & c \end{pmatrix}$$

Again, all the rows are linearly independent. However, we know that c is an exact sum of some h's. Our strategy is to find another basis of this lattice, consisting of a vector with a last element equal to 0. Moreover, since it can be represented as a linear combination of the vectors of the current basis, we know that this specific vector will only have elements equal to 0 or -1. A value of 0 on column i will give us feedback that h_i doesn't participate in the decomposition of c, while -1 indicate that h_i is used in the construction of c.

But how to find such basis? The following algorithm will help us:

Theorem 1 *(Lenstra, Lenstra, Lovász. [1] [2]) Let $L \in \mathbb{Z}^n$ be a lattice spanned by $B = \{v_1, ..., v_n\}$. The L^3-algorithm outputs a reduced lattice basis $\{v_1, ..., v_n\}$ with*

$$\|v_i\| \leq 2^{\frac{n(n-1)}{4(n-i+1)}} \, det(L)^{\frac{1}{n-i+1}} \quad for\ i = 1, ..., n \tag{8}$$

in time polynomial in n and in the bit-size of the entries of the basis matrix B.

In other words, the L^3 algorithm will yield another basis on the lattice, which consists of vectors with restrained norms defined in equation 8. The L^3 algorithm is already built-in *SageMath.*

Let's say that Eve intercepts a message between Alice and Bob, which is encrypted with Merkle-Hellman knapsack cryptosystem. Since everyone has access to the public key of the cryptosystem Eve possesses it too. The intercepted message C is:

$$\begin{aligned} C = [&318668, 317632, 226697, 388930, 357448, 297811, \\ &344670, 219717, 388930, 307414, 220516, 281175] \end{aligned}$$

The corresponding public key hard knapsack H is the vector:

$$H = [106507, 31482, 107518, 60659, 80717, 81516, 117973, 87697]$$

In order to recover the message Eve has to decrypt each element in C. For example, let's start with 318668. First, we need to initialize C and H:

```
sage: H = [106507, 31482, 107518, 60659,
             80717, 81516, 117973, 87697]
sage: c = 318668
```

Then, we start the construction of the lattice by first constructing the identity matrix:

```
sage: I = identity_matrix(8)
sage: I
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

We add another row full with zeroes:

```
sage: I = I.insert_row(8, [0 for x in range(8)])
sage: I
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Finally, we add the last column with H transposed and c. However, we will flip the sign of c – this way the first vector of the reduced basis should have a last element equal to 0 and all other elements equal to 1 (instead of -1).

```
sage: L_helper = [[x] for x in H]   # vector of vectors
sage: L_helper.append([-c])
sage: L = I.augment(matrix(L_helper))
sage: L
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 106507 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 31482 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 107518 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 60659 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 80717 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 81516 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 117973 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 87697 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -318668 \end{pmatrix}$$

Now, in order to reduce the basis, we will apply the L^3 algorithm[19] by just calling the *SageMath* `LLL()` function.

```
sage: L.LLL()
```

[19]See https://en.wikipedia.org/wiki/Lenstra%E2%80%93Lenstra%E2%80%93Lov%C3%A1sz_lattice_basis_reduction_algorithm

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ -1 & 1 & 0 & 1 & -1 & -2 & -2 & 2 & 1 \\ 3 & 1 & 2 & -1 & 1 & 1 & -1 & 1 & 1 \\ 1 & -1 & -2 & -1 & -3 & -1 & 1 & 1 & 1 \\ 2 & -2 & -1 & 1 & 0 & 2 & -3 & 1 & 1 \\ 0 & 0 & 3 & -4 & -2 & 1 & 0 & 0 & 0 \\ -1 & 3 & -1 & 3 & 0 & 0 & -1 & -3 & 2 \\ 0 & -1 & 1 & 4 & 0 & 0 & 0 & 0 & 4 \\ -2 & -1 & -2 & -3 & 1 & -1 & 2 & 1 & 3 \end{pmatrix}$$

The first candidate (the shortest vector in the reduced basis) is the one we were looking for:

```
sage: L.LLL()[0][:-1].dot_product(vector(H))
318668
```

So, the binary representation of the encrypted character is 01000111. By again using *SageMath*, we can easily check the corresponding letter:

```
sage: bs = ''.join([str(x) for x in L.LLL()[0][:-1]])
sage: bs
'01000111'

sage: chr(int(bs,2))
'G'
```

Challenge 9 *So,* ***G*** *is the first letter from the recovered text. By using* SageMath*, lattices and LLL algorithm, can you recover the remaining text?*

Screenshots from CT2 in section 15.2 on page 81 show a visualization using the mouse how to reduce a 2-dim basis with Gauss and a ready-to-run LLL implementation to reduce the basis of higher-dimensional bases.

10 Lattices and RSA

RSA is one of the earliest asymmetric cryptosystems. This section assumes that you are already familiar how the RSA cryptosystem works. However, we will briefly recall the basics of the key generation of the RSA algorithm.

10.1 Textbook RSA

- Two large distinct primes p and q are generated.
- Their product $n = pq$ is called the **modulus**.
- Then, we pick a number e, s.t. e is relatively prime to $\phi(n)$.[20] We define e as the **public key exponent**.
- We calculate d as the modular multiplicative inverse of e modulo $\phi(n)$. We define d as the **private key exponent**.
- The pair (n, e) forms the public key.
- The pair (n, d) forms the private key.

In order to avoid some known attacks to RSA, we need to choose our parameters wisely. Some of the requirements and recommendations are available in [3].

Now, let's encrypt the word "asymmetric" using *SageMath* and the RSA cryptosystem. First, we need to think of encoding strategy, i.e. translating strings to numbers. Throughout this section, we will use the following encoding procedure:

- Let's denote the string to be encoded as $S = s_1 s_2 \cdots s_n$.
- We replace each symbol s_i in the string to it's decimal ascii code representation. For example, the symbol "g" is replaced with "103".
- Then, each decimal ascii code is replaced with it's binary representation. For reversibility purposes, while the length of the binary representation is less than 8, we append at the beginning as many 0 as needed. For example, the binary representation of "103" is "1100111". However, the length of the binary representation is 7, so we append one more zero at the beginning to get "01100111".
- Finally, we convert S to decimal integer.

[20] $\phi(n)$ denotes the Euler's totient function. Since p and q are primes, we have $\phi(n) = (p-1)(q-1)$

For example, let's encode the word "asymmetric". First, we replace each symbol of S with its corresponding decimal ascii value:

```
sage: S = "asymmetric"
sage: S_ascii = [ord(x) for x in S]
sage: S_ascii
[97, 115, 121, 109, 109, 101, 116, 114, 105, 99]
```

Now, we continue by replacing each element in S_ascii by it's binary-padded equivalent:

```
sage: S_bin = [bin(x)[2:].zfill(8) for x in S_ascii]
sage: S_bin
```

$$\begin{gathered}01100001, \quad 01110011, \quad 01111001, \quad 01101101, \quad 01101101, \\ 01100101, \quad 01110100, \quad 01110010, \quad 01101001, \quad 01100011\end{gathered}$$

Finally, we concatenate all the elements in S_bin and convert it to a decimal number:

```
sage: SS = Integer(''.join(S_bin),2)
sage: SS
460199674176765747685731
```

For checking the reversibility of the encoding procedure, let's decode the result back:

```
sage: SS_bin = bin(SS)[2:]
sage: while len(SS_bin) % 8 != 0:
....:     SS_bin = '0' + SS_bin
sage: SS_ascii = [chr(int(SS_bin[x*8:8*(x+1)],2))
....:      for x in range(len(SS_bin)/8)]
sage: ''.join(SS_ascii)
'asymmetric'
```

When we are ready with the encoding procedure, we initialize the RSA parameters generation step. It's time to generate p, q and n:

```
sage: b = 512
sage: p = random_prime(2**b-1, lbound=2**(b-1)+2**(b-2))
```

With the previous example we generated a random prime number contained in the interval $\left[2^{b-1}+2^{b-2}, 2^b-1\right]$. Let's say we have two prime numbers in this interval, i.e.:

$$p = 2^{b-1} + 2^{b-2} + \rho_1$$

$$q = 2^{b-1} + 2^{b-2} + \rho_2$$

for some ρ_1 and ρ_2. Then, when we multiply them we will have:

$$\begin{aligned}
p \cdot q = (2^{b-1} + 2^{b-2} + \rho_1)(2^{b-1} + 2^{b-2} + \rho_2) = \\
&= 2^{2b-2} + 2^{2b-3} + 2^{b-1}\rho_2 + 2^{2b-3} + 2^{2b-4} + \\
&+ 2^{b-2}\rho_2 + \rho_1 2^{b-1} + \rho_1 2^{b-2} + \rho_1\rho_2 = \\
&= 2^{2b-2} + 2 \cdot 2^{2b-3} + \Omega = \\
&= 2^{2b-2} + 2^{2b-2} + \Omega = \\
&= 2 \cdot 2^{2b-2} + \Omega = 2^{2b-1} + \Omega > 2^{2b-1}
\end{aligned}$$

This guarantees that the bit length of their multiplication is $2b$.

```
sage: p.nbits()
512
```

The method `nbits()` returns the bit length of a number.

```
sage: q = random_prime(2**b-1, lbound=2**(b-1)+2**(b-2))
sage: N = p*q
sage: N.nbits()
1024
```

It's time to choose the public exponent e. Common choice of value of e is $2^{16} + 1$.

```
sage: e = 2**16 + 1
sage: e
65537
```

SageMath has a built-in function `euler_phi()`. However, if we directly type `euler_phi(N)`, *SageMath* will try to factor N.

```
sage: phi_N = (p-1)*(q-1)
```

Now, having $\phi(n)$ we can calculate d by using the built-in function `inverse_mod()`:

```
sage: d = inverse_mod(e,phi_N)
```

Let's assure that $ed \equiv 1 \mod \phi(n)$:

```
sage: e*d % phi_N
1
```

We are ready to encrypt the encoding `SS` of the message "asymmetric". The encryption can be directly calculated by using `SS**e%N`. However, we will use the built-in function `power_mod()`, which is considerably faster than the direct calculation.

```
sage: encrypted = power_mod(SS,e,N)
```

In order to decrypt the message:

```
sage: decrypted = power_mod(encrypted,d,N)
sage: decrypted
460199674176765747685731
```

Challenge 10 *Alice and Bob again decided to use their own encoding scheme and RSA implementation to secure their communication. The used encoding scheme is pretty straightforward – Alice translates each letter from the plaintext to its decimal ascii representation. In this way, Alice sends as many encrypted messages as the length of the original unencrypted message.*

This scheme has one major drawback – when large plaintexts are encrypted, the collection of the intercepted encrypted messages are vulnerable to frequency analysis attacks. To get around this, Alice and Bob renew their RSA keys when a threshold of sent messages is reached.

However, they made another mistake. The RSA public key is:

N = 68421763258426820318471259682647346897299270457991365227523187215179279
93776878211790146955615938091152726743120686152933384202585716854144646470442
80508081145003017193806309189089357804891172726923520981641104138226426702986
57847312225801755784399864594975116547815856011474793429956418177277806187836
101061

$e = 127$

Eve intercepted the following 11 messages (sent in this order):

$c1$ = 20842273115788965044434568911351496365025941709963878891654635864614247
25059541533787767041288429738064580955622451305616498186131307715129465784375
45536576877295737412743269079289912212463717642252956693458647654491632543974
2396928355234712078183

$c2$ = 20893454506753506195646780042379588087816651548061669824679147298111722
46521053196269736493675888244684173898988722302290793814087306806212626014865
44038910178129194905885355015942356215299224086980857408596226394217331686336
2277246322130097359903249313

$c3$ = 15351988337964905099701960376347076532439043589617467841763029200910162
25867008339499789389756629361807066970996087640185785892959509059061040916309

8252906678940270348578645367087079347216129233790173543224953780408040213693490500724979084761978285646999197843456

$c4$ = 27904208858505919506689042950498726643957813119400643293322771534900822430923585848781848455603612081226575100801226570484212026594858252089217840328837906708276016306114842897236574701434742246311142664328247890170520592851161647470983489359620795002699

$c5$ = 14562438053393942865563565506076319964337086564418653275680839364685346358348263872708128968423412681687735816462730409112745256517215618953897227627256898533454858045297931958376394955610471867756244498725191655684274134657700794939801031701760045360349184

$c6$ = 37370666535363066168961624931547694539547602092327751979114535130065929115448532953082477972777170290304404725670126936586698604529648793581659263060970546938259944838952911170478265448614822495177677220252704340545251785434955476627944717241828329

$c7$ = 57018830572739461491984788995673407709179639540390781558806685226845173001582252740946299992152591496992831944316269907785235915676185879264232465783672876342034636885982343764812696958235155060812119686263202672834115657789006658553081283546825372990992701071

$c8$ = 45667206025566800148122417818312587397117202643844088800399634507595677539812531804389800633373563635203026530295808267186537869501854999997585813165610459945099323041449890076258008953903360989998622098817497527261455918497690247104725594122565082035057621175773

$c9$ = 1686227313539318647886505059139335446746996624220331975878146712709645794810888946761963350628222465151134851313061316471360362284419753231478405415985364477239725795743107788714671289354822510203766481055710075778057712258940862586529599570943303841410139029504

$c10$ = 341849165158053526832505863192782931224162822759788612832891748659486026106737910383076021602844922593096922368623753010447251025423582399396141954459468286768770464472831982875580635659918156592765109749350304246573018358473129678989

$c11$ = 5112410579669852183534608382871115572785704026088086397778432135886054190982581109427995967742224987294310956529550255351980372648223511404048486808382051821395722995189698471430744248012819713379428438493366462166096818135752055667353488388471305370330810546875

Can you decrypt each of these messages and reconstruct the original message?

Screenshots from CT1 about a ready-to-run implementation of an attack against textbook RSA can be found in section 15.1 on page 77.

10.2 Lattices versus RSA

In [4] and [5] a whole new family of attacks on RSA are published – attacks which are using lattices and lattice reduction algorithms.

As we showed earlier, it is easy to to find the roots of a polynomial in a single variable over the integers. However, finding the roots of a **modular** polynomial is hard, i.e:

$$f(x) \equiv 0 \mod N$$

Let's denote N as a large composite integer of unknown factorization, and let's have an univariate integer polynomial (a polynomial in a single variable) $f(x)$ with degree n, i.e:

$$f(x) = x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 n + a_0$$

Furthermore, suppose there is an integer solution x_0 for the modular equation $f(x) \equiv 0 \mod N$, s.t. $x_0 < N^{\frac{1}{n}}$. D. Coppersmith showed how we can recover this value in polynomial time by using the following theorem of Howgrave-Graham:

Theorem 2 *Let $g(x)$ be an univariate polynomial with n monomials (a polynomial with just one term) and m be a positive integer. If we have some restraint X and the following equations hold:*

$$g(x_0) \equiv 0 \mod N^m, |x_0| \leq X \tag{9}$$

$$||g(xX)|| < \frac{N^m}{\sqrt{n}} \tag{10}$$

then $g(x_0) = 0$ holds over the integers.

The reasoning to use lattices:

- If we have some polynomials, which share the same root x_0 over N^m, we can represent each of them as a row from a lattice. Then, each linear combination of rows from the lattice will yield another polynomial having a root x_0.
- Then, by using the LLL algorithm on the specially designed lattice, in polynomial time we can find another reduced lattice basis, such that the norm of the shortest vector from the reduces basis will successfully satisfy the inequality in 10.
- Let's define the shortest vector in the reduces basis as $v = (v_0, v_1, \cdots, v_n)$. We construct the polynomial $g(x)$, s.t.:

$$g(x) = v_0 + \frac{v_1}{X}x + \frac{v_2}{X^2}x^2 + \cdots + \frac{v_n}{X^n}x^n$$

Since $g(x)$ is on the lattice, we know that

$$\begin{aligned} g(x_0) &\equiv 0 \mod N^m \\ |x_0| &\leq X \\ \deg(g) &= n \\ ||g(xX)|| &< \frac{N^m}{\sqrt{n+1}} \end{aligned}$$

Following the results from theorem 2 we can conclude that $g(x) = 0$ holds over the integers.

We can easily create polynomials sharing the same root x_0 over N^m. Consider the family of polynomials $g_{i,j}(x)$, s.t:

$$\begin{aligned} g_{i,j}(x) &= x^j N^{m-i} f^i(x) \\ 0 &\leq i < m \\ 0 &\leq j < \deg(f) \end{aligned}$$

By design, they all share the same root x_0 over N^m, i.e. $g_{i,j}(x_0) \equiv 0 \mod N^m$. The greater the value of m is, the more polynomials we can construct. The more polynomials we constructed, the bigger the lattice is and the greater the time to reduce the lattice will be.

Now, imagine Eve intercepted a set of messages in plaintext between Alice and Bob. The messages were:

The password for AES usage is: 4{8dXY!
The password for AES usage is: 31kTbwj
The password for AES usage is: 2rr#ETh
...
The password for AES usage is: &H,45zU

Then, Alice and Bob start exchanging files encrypted with AES using the communicated password. If new password is received, they immediately start using it. However, they realized that this is totally insecure and increased their security by using RSA.

They were using the same encoding procedure demonstrated at the beginning in this section. As we showed, the word **asymmetric** is encoded to the decimal number **460199674176765747685731**.

Let's say Alice wants to send an RSA encrypted string message s to Bob. She first encodes it to the decimal integer m. Then, she encrypts it by using Bob public key (N, e), i.e. $c = (m^e) \mod N$ and sends the encrypted message c through the insecure channel. Bob recovers the original message by using his private exponent, i.e. $c^d = m \mod N$.

The public key of Bob has parameters $(N, 3)$, where the bit length of N is 512. The predictive nature of the message (popular as stereotyped messages) can lead to devastating attack. Eve knows that the structure of the string message S' is:

$$S' = \text{"The password for AES usage is: } C_1C_2 \cdots C_7\text{"}$$

for some characters C_i. Before encrypting, Alice has to translate each character to its ascii binary string representation. Let's denote the binary translating function as $T_1(A)$ for some character or string A, i.e. $T_1(\text{"}xy\text{"}) = T_1(\text{"}x\text{"})||T_1(\text{"}y\text{"})$, where symbol $||$ denotes the symbol for concatenation of strings.

Having this in mind, we can write down:

$T_1(S') = T_1(\text{"The password for AES usage is: "})||T_1(\text{"}C_1C_2 \cdots C_7\text{"})$

After this translation, Alice reads the final binary string as a decimal number. Let's denote this function as $T_2(S')$.

Each ascii decimal representation of C_i is in the interval $[0, 255]$. Let's denote as C_{00} the symbol with ascii decimal representation 0, and as C_{ff} the symbol with ascii decimal representation 255. For simplicity, let's denote

$$B = \text{"The password for AES usage is: "}$$

Having the encoding procedure in mind, we can conclude that:

$$\begin{gathered} T_2(T_1(B||C_{00}C_{00} \cdots C_{00})) < \\ < T_2(T_1(B||C_1C_2 \cdots C_7)) < \\ < T_2(T_1(B||C_{ff}C_{ff} \cdots C_{ff})) \end{gathered}$$

Let's introduce two new variables: a and X, s.t.:

$$\begin{aligned} a &= T_2(T_1(B||C_{00}C_{00} \cdots C_{00})) \\ X &= T_2(T_1(C_{ff}C_{ff} \cdots C_{ff})) \end{aligned}$$

We are looking for such x, s.t.

$$(a + x)^3 \equiv c \mod N$$

or

$$(a + x)^3 - c \equiv 0 \mod N$$

In fact, x is denoting the difference between $T_2(T_1(C_1C_2 \cdots C_7))$ and $T_2(T_1(C_{00}C_{00} \cdots C_{00}))$.

Let's stop for a while and implement the current polynomial by using *SageMath*. First, we introduce the function `encode()` – it is an equivalent to $T_2(T_1(m))$. Here is a sample how to call this function and what it outputs: encode("A"): 65, encode("AB"): 16706, encode("ABC"): 4276803.

```
sage: def encode(m):
....:    return Integer(''.join([bin(ord(x))[2:].
....:    zfill(8) for x in m]),2)
```

We introduce the expected starting characters of the encrypted message.

```
sage: B = "The password for AES usage is: "
```

Now, we plug-in the values of $C_{00}C_{00} \cdots C_{00}$ and $C_{ff}C_{ff} \cdots C_{ff}$.

```
sage: padding = ''.join(['\x00' for x in range(7)])
sage: X_str = ''.join(['\xff' for x in range(7)])
```

We continue by calculating the values of a and X:

```
sage: a_str = B + padding
sage: a_const = encode(a_str)
sage: X_const = encode(X_str)
```

In order to illustrate the attack better, we will construct a multivariate polynomial over the integer ring, instead of a univariate.

```
sage: R.<X,N,a,c> = ZZ[]
```

Now, we are ready to construct the polynomial $f(X)$:

```
sage: f = (X+a)**3 - c
sage: f
X^3 + 3*X^2*a + 3*X*a^2 + a^3 - c
```

We don't know x_0. However, we know a good upper limit for x_0, i.e. $x_0 < X$. Since $e = 3$, the degree of our polynomial is 3. For this particular case, let's fix m with the smallest possible value, i.e. $m = 1$:

```
sage: f.degree()
3
sage: m = 1
```

Our lattice will be with dimension 4 - we have exactly 3 polynomials $g_{i,j}$, as well as the final polynomial f.

```
sage: dim = 4
sage: M = matrix(dim,dim)
```

Now, we start the construction of the polynomials accordingly. Following the strategy of the construction of the lattice, we need to define the following lattice:

$$\begin{pmatrix} N & 0 & 0 & 0 \\ 0 & NX & 0 & 0 \\ 0 & 0 & NX^2 & 0 \\ a^3 - c & 3a^2X & 3aX^2 & X^3 \end{pmatrix}$$

For achieving this, we first need to define the helper function `get_ext_monoms()`. It will help us to extract all the monomials from a given polynomial, but with the coefficients included.

```
sage: def get_ext_monoms(ff):
....:     ff_m = ff.monomials()
....:     ff_coefs = [ff.monomial_coefficient(x) for x in ff_m]
....:     ff_monoms_ext = [ff_m[x]*ff_coefs[x]
....:     for x in range(len(ff_m))]
....:     return ff_monoms_ext
```

For example:

```
sage: get_ext_monoms(f)
[X^3, 3*X^2*a, 3*X*a^2, a^3, -c]
```

However, there is an issue here – a^3 and c are treated as separated monomials. That's why we should apply a final substitution right before calling the function `get_ext_monoms()`:

```
sage: for i in range(m):
....:    for j in range(e):
....:      g = X**j * N**(m-i) * (f**i)
....:      g = g.subs({N:N_const, a:a_const})
....:      g_monoms_ext = get_ext_monoms(g)
....:      for monom in g_monoms_ext:
....:        row_pos = e*i+j
....:        column_pos = monom.degree()
....:        M[row_pos,column_pos] = monom.subs({X:X_const})
```

Finally, we append the final row of the lattice – the polynomial f itself:

```
sage: fg = f**m
sage: fg = fg.subs({N:N_const, a:a_const})
sage: fg_monoms_ext = get_ext_monoms(fg)
sage: for fg_monom in fg_monoms_ext:
....:     pos = fg_monom.degree()
....:     M[dim-1, pos] = fg_monom.subs({X:X_const})
```

Our lattice is ready. We can initiate the lattice reduction algorithm:

```
sage: B = M.LLL()
```

The shortest vector `B[0]` in our reduced basis holds the coefficients we need in order to construct the polynomial g over the rational ring. We can easily construct it using *SageMath*:

```
sage: R.<x> = QQ[]
sage: Q = sum([B[0][i]*(x**i)/(X_const**i) for i in range(dim)])
```

Following theorem 2 the last polynomial should have a solution over the integers. And indeed:

```
sage: sol = Q.roots(ring=ZZ)[0][0]
sage: type(sol)
<type 'sage.rings.integer.Integer'>
```

Now, let's define another helper function `decode()`, which will translate back an encoded message Z, i.e. $T_1^{-1}(T_2^{-1}(Z))$:

```
sage: def decode(n):
....:   nn = str(bin(n)[2:])
....:   while len(nn) % 8 != 0:
....:     nn = '0' + nn
....:     return ''.join([chr(int(nn[x*8:8*(x+1)],2))
....:     for x in range(len(nn)/8)])
```

Our final step is to just decode the solution:

```
sage: decode(sol)
```

Challenge 11 *Eve inspected the public key of Bob and intercepted the encrypted message c:*

N = 871052631206654885022767148076180050911675472954145224034038582604459379782025841959769270115412869697265035907671892366767420776463584582195941126276 0499

$e = 3$

c = 533247982598794633956287465571096863623160823801198491330126244714226132257522454937130556627216506112493046973324957750347628241445331227809291995164455

As an exercise, can you recover the original message by using the lattice attack provided above?

Challenge 12 *Soon after Eve's successful lattice-reduction attack, Alice and Bob had been aware of the insecure scheme they are using to encrypt their correspondence. However, they thought that this was possible because they were using too short passwords. They increased the passwords from a length of* ***7*** *to a length of* ***13****.*

The public key of Bob was left intact. The newly intercepted message c is:

c = 74875898628081924295230958863232737993265641657662214847638483603488006886812559351998510022388544254768721487918947883514843899598628979968735140562 82948

Can you recover the newly exchanged password? Here are some things to consider:

- *First let's try with $m = 1$. Why the attack fails?*
- *Now, let's try with $m = 2$. The dimension of the new lattice will be $em + 1$.*

- *The polynomial f is always the same. However, we need more helper polynomials $g_{i,j}$ in order to construct our bigger lattice. At then end, you should construct the following lattice:*

$$\begin{pmatrix} N^2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & N^2X & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & N^2X^2 & 0 & 0 & 0 & 0 \\ N(a^3-c) & 3Na^2X & 3NaX^2 & NX^3 & 0 & 0 & 0 \\ 0 & N(a^3-c)X & 3Na^2X^2 & 3NaX^3 & NX^4 & 0 & 0 \\ 0 & 0 & N(a^3-c)X^2 & 3Na^2X^3 & 3NaX^4 & NX^5 & 0 \\ f_0 & f_1 & f_2 & f_3 & 15a^2X^4 & 6aX^5 & X^6 \end{pmatrix}$$

$$\begin{aligned} f_0 &= (a^3-c)^2 \\ f_1 &= 6a^2(a^3-c)X \\ f_2 &= a(15a^3-6c)X^2 \\ f_3 &= (20a^3-2c)X^3 \end{aligned}$$

Challenge 13 *For the next stereotype challenge, you have the following parameters:*

N = 11225354852531229312705821542018938144842129865964887302659527454109100 72681138663483074618935128265451387560973724847297085037894275160093985827338 65515455177790394159554613094757808985408328307994023228782530102763869568783 56093590307746836948987210933431011897995020707110828021962036273746776030822 7448837

$e = 7$

c = 10670654096244930306696108877164811975817784621106090830133614424028968837154232320341636292740214826278191136787096724376919541317293439292857379222 72207153114174438757138142540189592431327528606195874021248932484514678389202 73794758310827554392841573679450441556883666302722319029010463140829183505391 092171

*This time, the degree of the polynomial f is **7**. You need to try different values of m in order to construct a large enough, but still compact lattice.*

Screenshots from CT1 and CT2 about ready-to-run lattice-based implementations of attacks against RSA can be found in section 15.1 on page 77 and in section 15.2 on page 81.

11 Lattice Basis Reduction

A given lattice has infinitely many different bases. The main goal of lattice basis reduction is to find (by using some lattice basis as an input) a basis that consists of short vectors, or, equivalently, a basis consisting of vectors that are pairwise nearly orthogonal. Thus, the reduced basis may lead to a solution of an underlying problem, like breaking a knapsack cryptosystem, as we have already shown in section 9.2 on page 39.

Let's first introduce the notion of Gram-Schmidt orthogonalization named after the mathematicians Jørgen Pedersen Gram and Erhard Schmidt.

Definition 26 *With an ordered lattice basis* $b_1, \cdots, b_m \in \mathbb{R}^n$ *we associate the Gram-Schmidt orthogonalization* $\hat{b}_1, ..., \hat{b}_m \in \mathbb{R}^n$ *which can be computed from* $b_1, \cdots, b_m$ *together with the Gram-Schmidt coefficients* $\mu_{i,j} = \frac{b_i \cdot \hat{b}_j}{\hat{b}_j \cdot \hat{b}_j}$ *by the recursion* $\hat{b}_1 = b_1, \hat{b}_i = b_i - \sum_{j=1}^{i-1} \mu_{i,j} \hat{b}_j$ *for* $i = 2, ..., m$.
Let $\operatorname{span}(b_1, ..., b_{i-1})^\perp$ *be the set of all vectors orthogonal to* $\operatorname{span}(b_1, ..., b_{i-1})$, *i.e.*

$$\operatorname{span}(b_1, ..., b_{i-1})^\perp = \{v \in \mathbb{R}^n | v \cdot \sum_{j=1}^{i-1} x_j b_j = 0 \ \forall x_j \in \mathbb{R}\}.$$

The orthogonal projections of vectors b_j *to* $\operatorname{span}(b_1, ..., b_{i-1})^\perp$ *are named* π_i
$\pi_i : \mathbb{R}^n \to \operatorname{span}(b_1, ..., b_{i-1})^\perp, \quad \pi_i(b_j) := \sum_{t=i}^{j} \mu_{j,t} \hat{b}_t, \quad i = 1, \cdots, m.$

We have $\mu_{i,i} = 1$ and $\mu_{i,j} = 0$ for $i < j$. If the basis $b_1, \cdots, b_m$ is integral, then the vectors $\hat{b}_1, \cdots, \hat{b}_m$ and the Gram-Schmidt coefficients are rational. We can write the above equations in matrix notation as:

$$(b_1, \cdots, b_m) = (\hat{b}_1, \cdots, \hat{b}_m)(\mu_{i,j})^T_{1 \le i,j \le m}$$

Definition 27 *The i-th successive minimum* λ_i *of a lattice* L *is defined as the minimum radius* $r \in \mathbb{R}$ *of a n-dimensional sphere* B *with center 0 that contains* i *linearly independent lattice vectors:*

$$\lambda_i(L) = \min\{r | dim(spanL \cap B_{r,0}) \ge i\}.$$

Obviously λ_1 is the norm of the shortest nonzero lattice vector.

One of the strongest notions of lattice reduction is based on the work of Hermite [7], Korkine and Zolotarev [8, 9, 10]:

Definition 28 *A lattice basis $b_1, ..., b_m$ is called reduced in the sense of Hermite, Korkine and Zolotarev or short HKZ-reduced if the following holds:*
1. $|\mu_{i,j}| \leq \frac{1}{2}$ *for* $1 \leq j < i \leq m$,
2. $||\hat{b}_i|| = \lambda_1(L(\pi_i(b_i), ..., \pi_i(b_m))$ *for* $1 \leq i \leq m$.

The first vector of any HKZ-reduced lattice basis is a shortest lattice vector.

Let's take a look at two-dimensional lattices where we can easily find a shortest lattice vector with respect to the Euclidean norm by using the *Gauss reduction* algorithm. The process is similar to the process of calculating the greatest common divisior of two integers by applying the Euclidean algorithm.

```
Input: lattice basis $\{a, b\}$
REPEAT
    $\{a, b\} = \{b - \left\lceil \frac{a \cdot b}{a \cdot a} \right\rfloor * a, a\}$
UNTIL $||a|| \leq ||b|| \leq ||a - b||$
Output: Gauss reduced lattice basis $\{a, b\}$
```

For any real number x, $\lceil x \rfloor$ denotes the closest integer, i.e. $\lceil x \rfloor = \lfloor x + 0.5 \rfloor$.

Example 13 *Let's run the Gauss reduction algorithm on a basis* $B = \{a, b\} = \{(1, 7), (-1, 1)\}$ *of a given lattice* L *in* $\mathbb{Z}^2$.

Input: *Lattice basis* $\{a, b\} = \{(1, 7), (-1, 1)\}$

$$\begin{aligned}\{a, b\} &= \{b - \left\lceil \frac{a \cdot b}{a \cdot a} \right\rfloor \cdot a, a\} \\ &= \{(-1, 1) - \left\lceil \frac{(1, 7) \cdot (-1, 1)}{(1, 7) \cdot (1, 7)} \right\rfloor \cdot (1, 7), (1, 7)\} \\ &= \{(-1, 1) - \left\lceil \frac{6}{50} \right\rfloor \cdot (1, 7), (1, 7)\} \\ &= \{(-1, 1), (1, 7)\}\end{aligned}$$

Since $||b|| = \sqrt{50} > \sqrt{40} = ||a - b||$ *we need to run another iteration:*

$$\begin{aligned}
\{a,b\} &= \{b - \left\lceil \frac{a \cdot b}{a \cdot a} \right\rfloor \cdot a, a\} \\
&= \{(1,7) - \left\lceil \frac{(-1,1)\cdot(1,7)}{(-1,1)\cdot(-1,1)} \right\rfloor \cdot (-1,1), (-1,1)\} \\
&= \{(1,7) - \left\lceil \frac{6}{2} \right\rfloor \cdot (-1,1), (-1,1)\} \\
&= \{(1,7) - 3 \cdot (-1,1), (-1,1)\} \\
&= \{(4,4), (-1,1)\}
\end{aligned}$$

Now $||a|| = \sqrt{32} > \sqrt{2} = ||b||$ *and we need another iteration:*

$$\begin{aligned}
\{a,b\} &= \{b - \left\lceil \frac{a \cdot b}{a \cdot a} \right\rfloor \cdot a, a\} \\
&= \{(-1,1) - \left\lceil \frac{(4,4)\cdot(-1,1)}{(4,4)\cdot(4,4)} \right\rfloor \cdot (4,4), (4,4)\} \\
&= \{(-1,1) - \left\lceil \frac{0}{32} \right\rfloor \cdot (-1,1), (4,4)\} \\
&= \{(-1,1), (4,4)\}
\end{aligned}$$

Since $||a|| = \sqrt{2} < \sqrt{32} = ||b|| < \sqrt{34} = ||a-b||$*, the algorithm ends.*

Output: $\{a,b\} = \{(-1,1),(4,4)\}$

$(-1,1)$ *is a shortest non-zero vector in* L *and* $(4,4)$ *is a shortest vector of* L *that is linear independent from* $(-1,1)$

In hgher dimensions the calculation of an HKZ-reduced bases (see page 58) is very inefficient (no polynomial time algorithm is known) we define a hierarchy of notions that approximate HKZ-reduced bases in reasonable time.

Definition 29 *An ordered lattice basis* $b_1, \cdots, b_m \in \mathbb{R}^n$ *is called* size-reduced *if* $|\mu_{i,j}| \leq \frac{1}{2}$ *for* $1 \leq j < i \leq m$. *An individual basis vector* b_i *is* size-reduced *if* $|\mu_{i,j}| \leq \frac{1}{2}$ *for* $1 \leq j < i$.

A size reduced lattice basis consists of vectors that are almost orthogonal to each other.

In 9.2 we already used the LLL algorithm in *SageMath* in order reduce a basis and solve a knapsack (see 41). We now give a formal definition of LLL-reduction as well as the details of the algorithm as this is used as a subroutine in further algorithms described later in this chapter.

Definition 30 *Let δ be a constant, $0 < \delta \leq 1$. We call a basis $b_1, ..., b_m \in \mathbb{R}^n$* LLL-reduced with δ *if it is size-reduced, s.t.*

$$\delta ||\hat{b}_{k-1}||^2 \leq ||\hat{b}_k + \mu_{k,k-1}\hat{b}_{k-1}||^2 \textit{ for } k = 2, \cdots, m$$

The two algorithms for calculating respectively the ordered lattice basis and LLL-reduced δ basis, are summarized below.

```
Algorithm for size reduction of basis vector $b_k$
Input:   $b_1, ..., b_m \in \mathbb{R}^n$ (lattice basis)
         $\mu_{i,j}$ for $1 \leq j < i \leq m$ its Gram-Schmidt coefficients)
         FOR $j = k-1, ..., 1$
               IF $|\mu_{k,j}| > \frac{1}{2}$ THEN
                  $b_k := b_k - \lceil \mu_{k,j} \rfloor b_j$
                  FOR $i = 1, ..., k-1$ DO $\mu_{k,i} := \mu_{k,i} - \lceil \mu_{k,j} \rfloor \mu_{j,i}$
               END IF
         END FOR $j$
Output:  $b_1, ..., b_m \in \mathbb{R}^n$ (lattice basis where $b_k$ is size-reduced)
         $\mu_{i,j}$ for $1 \leq j < i \leq m$ (its Gram-Schmidt coefficients)
```

Algorithm for LLL reduction

```
Input:   $b_1, ..., b_m \in \mathbb{R}^n$ (lattice basis), $\delta$ with $0 < \delta \leq 1$.
1. $k := 2$ (k is the stage; when entering stage k, the basis $b_1, ..., b_{k-1}$
            is already LLL-reduced with $\delta$, the Gram-Schmidt coefficients
            $\mu_{i,j}$ are calculated for $1 \leq j < i < k$
            as well as the normsquares $c_i = ||\hat{b}_i||_2^2$ for $i = 1, ..., k-1$)
2. WHILE $k \leq m$
           FOR $j = 1, ..., k-1$
                 $\mu_{k,j} := (b_k \cdot b_j - \sum_{i=1}^{j-1} \mu_{j,i}\mu_{k,i}c_i)/c_j$
           $c_k := b_k \cdot b_k - \sum_{j=1}^{k-1} \mu_{k,j}c_j$
3.         (size-reduce $b_k$)
           FOR $j = k-1, ..., 1$
                 $\mu := \lceil \mu_{k,j} \rfloor$
                 FOR $i = 1, ..., j-1$
                      $\mu_{k,i} := \mu_{k,i} - \mu\mu_{j,i}$
                 $\mu_{k,j} := \mu_{k,j} - \mu$
                 $b_k := b_k - \mu b_j$
4.          IF $\delta c_{k-1} > c_k + \mu_{k,k-1}^2 c_{k-1}$
            THEN exchange $b_k$ and $b_{k-1}$
                 $k := max(k-1, 2)$
            ELSE $k := k+1$
   END WHILE
Output: basis $b_1, ..., b_m$ which is LLL-reduced with $\delta$
           Gram-Schmidt coefficients $\mu_{i,j}$ for $1 \leq j < i \leq m$
           normsquares $c_i = ||\hat{b}_i||_2^2$ for $i = 1, ..., m$
```

In practice, replacement of step 4 with the *deep insertion rule* proposed by Schnorr and Euchner [11] proofed to be more efficient (by still being polynomial in n, m and input length) for any fixed value t:

```
4. ($t$ deep insertions)
            $c := ||b_k||_2^2, T :=\min(t, k-1), i := 1$
            WHILE $i < T$
                  IF $\delta c_i > c$
                  THEN $(b_1, ..., b_k) := (b_1, ..., b_{i-1}, b_k, b_i, ..., b_{k-1})$
                       $k := \max(i-1, 2)$
                       GOTO 2.
                  ELSE $c := c - \mu_{k,i}^2 c_i$
                       $i := i+1$
            END WHILE
            IF $\delta c_{k-1} > c_k + \mu_{k,k-1}^2 c_{k-1}$
            THEN exchange $b_k$ and $b_{k-1}$
                  $k := max(k-1, 2)$
            ELSE $k := k+1$
```

Furthermore, Schnorr and Euchner [11] invented the notion of *blockwise Korine Zolotarev reduced bases* with block size β. For a lattice basis $b_1, ..., b_m$ and $\beta = 2$ the notion is equivalent to LLL-reduced basis and it is equivalent to the notion of HKZ reduced bases for $\beta = m$.

Definition 31 *Let $\beta \geq 2$ be an integer and $\delta \in (0, 1]$ be real. A basis $b_1, ..., b_m$ of a lattice $L \subset \mathbb{R}^n$ is called $(\beta, \delta)-$ block reduced*[21]*, if the following holds for $i = 1, ..., m$:*
1. $|\mu_{i,j}| \leq \frac{1}{2}$ for all $j < i$,
2. $\delta||\hat{b}_i||^2 \leq \lambda_1^2(L(\pi_i(b_i), ..., \pi_i(b_{min(i+\beta-1,m)})))$.

Although there is no proven polynomial bound for the number of operations of any algorithm to calculate a (β, δ)- block reduced basis for $\beta > 2$ (except for $\beta = 3$ and $\delta \in [\frac{1}{2}, \frac{1}{2}\sqrt{3})$, see [13]) the following algorithm proved to be efficient in practice for small bock sizes ($\beta \leq 30$). Its core component is the enumeration algorithm ENUM(j,k) which finds an integer, non-zero minimum $(u_j, ..., u_k)$ of

$$c_j(\tilde{u}_j, ..., \tilde{u}_k) := ||\pi_j(\sum_{i=j}^{k} \tilde{u}_i b_i)||_2^2, \ (\tilde{u}_j, ..., \tilde{u}_k) \in \mathbb{Z}^{k-j+1} \tag{11}$$

Before going into the details of ENUM(j,k) let's have a look at the block reduction algorithm below. It cyclically iterates over all positions j, ensures that the basis is size-reduced and enforces $\delta||\hat{b}_j||^2 \leq \lambda_1^2(L(\pi_j(b_j), ..., \pi_j(b_{min(j+\beta-1,m)})))$ until this holds for all j:

[21]in the literature, also the notion of BKZ-reduced bases is used for block reduced bases

```
Algorithm for $(\beta, \delta)$ block reduction
Input: basis $b_1, ..., b_m \in \mathbb{R}^n$ of $L,\ \beta \in \mathbb{N},\ 2 \leq \beta \leq m,\ \delta \in \mathbb{R},\ 0 \leq \delta \leq 1$
1. LLL-reduce $b_1, ..., b_\beta,\ j := m,\ z := 0$
2. WHILE $z < m - 1$
        $j := j + 1$, IF $j = m$ THEN $j := 1$
        $k :=$min$(j + \beta - 1, m)$
        ENUM(j,k) (this enumeration outputs integer coefficients
                $(u_j, ..., u_k)$ of a lattice vector $b_j^{new} = \sum_{i=j}^{k} u_i b_i$ and
                $\bar{c}_j := ||\pi_j(b_j^{new})||^2 = \lambda_1^2(L(\pi_j(b_j), ..., \pi_j(b_k))))$
        $h :=$min$(k + 1, m)$
        IF $\bar{c}_j < \delta c_j$
        THEN extend $b_1, ..., b_{j-1}, b_j^{new}$ to a basis
            $b_1, ..., b_{j-1}, b_j^{new}, ..., b_k^{new}, b_{k+1}, ..., b_m$ of L
            LLL-reduce $b_1, ..., b_h^{new}$
            $z := 0$
        ELSE LLL-reduce $b_1, ..., b_h$
            $z := z + 1$
   END WHILE
Output: $(\beta, \delta)$ block reduced basis $b_1, ..., b_m$
```

The extension of $b_1, ..., b_{j-1}, b_j^{new}$ to a basis $b_1, ..., b_{j-1}, b_j^{new}, ..., b_k^{new}, b_{k+1}, ..., b_m$ of L is done with the following algorithm:

```
Algorithm BASIS
Input: basis $b_1, ..., b_m, (u_j, ..., u_k)$
1. $b_j^{new} = \sum_{i=j}^{k} u_i b_i$
2. $g := \max\{t : j \leq t \leq k, u_t \neq 0\}$
3. WHILE $|u_g| > 1$
        $i := \max\{t : j \leq t < g : u_t \neq 0\}$
        $q := \lceil u_g / u_i \rfloor$
        $u_i := u_g - q \cdot u_i$
        $u_g := u_i^{old}$
        $b_g := q \cdot b_g + b_i$
        $b_i := b_g^{old}$
4. FOR $i = g, ..., j + 1$  $b_i := b_{i-1}$
5. $b_j := b_j^{new}$
Output: $b_1, ..., b_m$
```

By introducing the naming conventions $\tilde{c}_t := ||\pi_t(\sum_{i=t}^{k} \tilde{u}_i b_i||^2$ and $c_t := ||\hat{b}_t||^2 = ||\pi_t(b_t)||^2$, we get $\tilde{c}_t = \tilde{c}_{t+1} + (\tilde{u}_t + \sum_{i=t+1}^{k} \tilde{u}_i \mu_{i,t})^2 c_t$. For fixed $\tilde{u}_{t+1}, ..., \tilde{u}_k)$ we can easily enumerate all integers $\tilde{u}_t$, s.t. corresponding values of $\tilde{c}_t$ are non-decreasing, starting with $\tilde{u}_t = \left\lceil -\sum_{i=t+1}^{k} \tilde{u}_i \mu_{i,t} \right\rfloor$. The following (basic) variant of algorithm ENUM traverses the resulting search tree in depth-first search order. Other variants (e.g. traversing the tree in breadth-first search order or incomplete - pruned - traversals) are given in [13].

```
Algorithm ENUM(j,k)
Input: $j, k, c_i$ for $i = j, ..., k$ and $\mu_{i,t}$ for $j \leq t < i \leq k$
1. $s := t := j, \bar{c}_j := c_j, \tilde{u}_j := u_j := 1,$
   $v_j := y_j := \Delta_j := 0, \delta_j := 1,$
   FOR $i = j+1, ..., k+1$
         $\tilde{c}_i := u_i := \tilde{u}_i := v_i := y_i := \Delta_i := 0, \delta_i := 1$
2. WHILE $t \leq k$
         $\tilde{c}_t := \tilde{c}_{t+1} + (y_t + \tilde{u}_t)^2 c_t$
         IF $\tilde{c}_t < \bar{c}_j$
         THEN IF $t > j$
              THEN $t := t-1, y_t := \sum_{i=t+1}^{s} \tilde{u}_i \mu_{i,t},$
                   $\tilde{u}_t := v_t := \lceil -y_t \rfloor, \Delta_t := 0$
                   IF $\tilde{u}_t > -y_t$
                   THEN $\delta_t := -1$
                   ELSE $\delta_t := 1$
              ELSE $\bar{c}_j := \tilde{c}_j, u_i := \tilde{u}_i$ for $i = j, ..., k$
         ELSE $t := t+1$
              $s :=$max$(s, t)$
              IF $t < s$ THEN $\Delta_t := -\Delta_t$
              IF $\Delta_t \delta_t \geq 0$ THEN $\Delta_t := -\Delta_t + \delta_t$
              $\tilde{u}_t := v_t + \Delta_t$
   END WHILE
Output: $(u_j, ..., u_k), \bar{c}_j$
```

11.1 Breaking knapsack cryptosystems using lattice basis reduction algorithms

For given natural numbers $n, a_1, ..., a_n$ and s, a knapsack problem consists of either finding a vector $(x_1, ..., x_n) \in \{0, 1\}^n$ such that $\sum_{i=1}^{n} x_i a_i = s$ or to prove that no such solution exists. $x = (x_1, ..., x_n)$ is called a solution of the knapsack problem $(n, a_1, ..., a_n, s)$. As the corresponding decision problem is NP-complete, several cryptosystems based on the knapsack problem were proposed. In section 9.1 we already described and attacked the Merkle-Hellman knapsack cryptosystem. In the following we sketch the attacks on cryptosystems proposed by Chor and Rivest [14] and by Orton [15].

Breaking the Chor-Rivest cryptosystem

Chor and Rivest construct n special weights a_i and code a binary message $x = x_1...x_n$ with q 1s and $n-q$ 0s by $s := \sum_{i=1}^{n} x_i a_i$. Let's have a look at the following lattice basis B:

$$B := \begin{pmatrix} b_1 \\ \\ \vdots \\ \\ b_{n+1} \end{pmatrix} := \begin{pmatrix} 1 & q & q & ... & q & n^2 s & n^2 q \\ 0 & n & 0 & ... & 0 & n^2 a_1 & n^2 \\ 0 & 0 & n & & 0 & n^2 a_2 & n^2 \\ \vdots & & \ddots & \ddots & \vdots & \vdots & \vdots \\ 0 & ... & & & n & n^2 a_n & n^2 \end{pmatrix}$$

Any lattice vector $v = (v_0, ..., v_{n+2}) = \sum_{i=1}^{n+1} u_i b_i$ with $v_0 = \pm 1$ and $v_{n+1} = v_{n+2} = 0$ decodes the message x in case exactly q of the coefficients v_j have value $v_0 \cdot (q-n)$ and $n-q$ coefficients have value $v_0 \cdot q$. In this case we get

$$x_i = \begin{cases} 1, & \text{if} v_i = v_0 \cdot (q-n), \\ 0, & \text{if} v_i = v_0 \cdot q \end{cases}$$

In his Diploma thesis H. Hörner uses variants of the block reduction algorithm to find such vectors for parameters $(n, q) = (103, 12)$ and $(151, 16)$. Main results are published in [16]. We're not going to go into that here.

Breaking the Orton cryptosystem

In [15] Orton proposed the following public key cryptosystem based on so-called dense, compact and modular knapsack problems:

Public parameters: natural numbers r, n, s. (Messages consist of n blocks with s bits, r is the number of rounds to create the keys).

Secret key: integers $a_i^{(0)}$ with $a_1^{(0)} = 1, a_i^{(0)} > (2^s - 1) \sum_{j=1}^{i-1} a_j^{(0)}$ for $i = 1, ..., n$ and natural numbers $q_2, p^{(k)}, w^{(k)}$ for k=1,...,r, where $q_1 := p^{(r)}/q_2$ is an integer.

The part $\{a_i^{(0)}\}$ of the secret key represents an "easy" knapsack. It is transformed in a "hard" knapsack by the following transformations:

$$\begin{aligned} a_i^{(k)} &:= a_i^{(k-1)} w(k) \bmod p^{(k)} \text{ for } i = 1, ..., n+k-1,\ a_{n+k}^{(k)} := -p^{(k)}, \\ f_i^{(k)} &:= 2^{-\text{prec}(k)} \left\lfloor a_i^{(k)} 2^{\text{prec}(k)} / p^{(k)} \right\rfloor \text{ for } i = 1, ..., n+k-1,\ k = 1, ..., r, \\ a_{i,j} &:= a_i^{(r)} \bmod q_j \text{ for } i = 1, ..., n+r-1,\ j = 1, 2. \end{aligned}$$

The cryptosystem uses the secret "trapdoor" $q_2, p^{(k)}, w^{(k)}$ $(k = 1, ..., r)$. prec(k) is the number of precision bits for calculating the quotients $f_i^{(k)}$ in round k. Orton proposed to use $\text{prec}(k) = s + log_2 n + k + 2$ in order to ensure unique decryption and prevent attacks known before.

Public key: natural numbers $q_1, \text{prec}(k)$ for $k = 1, ..., r-1$
non-negative integers $a_{i,j}$ for $i = 1, ..., n+r-1,\ j = 1, 2$
rational numbers $f_i^{(k)} \in 2^{-\text{prec}(k)}[0, 2^{\text{prec}(k)})$ for $k = 1, ..., r-1,\ i = 1, ..., n+k-1$.

Encryption
Input: public key, message $(x_1, ..., x_n) \in [0, 2^s)^n$
1. $x_{n+k} := \left\lfloor \sum_{i=1}^{n+k-1} x_i f_i^{(k)} \right\rfloor$ for $k = 1, ..., r-1$
2. $y_1 := \sum_{i=1}^{n+r-1} x_i a_{i,1} \bmod q_1,\ y_2 := \sum_{i=1}^{n+r-1} x_i a_{i,2}$

Output: encrypted message (y_1, y_2)

Decryption
Input: public and secret key, encrypted message (y_1, y_2)
1. recombine $y^{(r)} \equiv y_j \bmod q_j$ (j=1,2) using Chinese remainder theorem:
   $y^{(r)} := q_2((y_1 - y_2)q_2^{-1} \bmod q_1) + y_2$
2. $y^{(k-1)} := y^{(k)}(w^{(k)})^{-1} \bmod p^{(k)}$ for $k = r, ..., 1$
3. solve $\sum_{i=1}^{n} x_i a_i^{(0)} = y^{(0)}$ with $x_i \in [0, 2^s)$.
   (this can easily be done since $a_i^{(0)} > (2^s - 1)\sum_{j=1}^{i-1} a_j^{(0)}$)

Output: decrypted message $(x_1, ..., x_n)$

In the following we show by using lattice algorithms how to break any message encrypted by the Orton cryptosystem. We first construct a lattice basis $b_1, ..., b_{m+2} \in \mathbb{Z}^{m+r+2}$ s.t. the original message can easily be recovered from lattice vector with l_∞-norm 1. The l_∞-norm $||v||_\infty$ of a vector $v = (v_1, ..., v_n)$ is defined the maximal absolute value of its coefficients v_i.

$$||v||_\infty = \max(|v_1|, \dots, |v_n|),\ v \in \mathbb{R}^n \tag{12}$$

We then show how such a lattice vector can be found efficiently.

The decryption problem is stated as follows: Given the public parameters (r, n, s) the public key $(q_1, \text{prec}(k), a_(, j, f_i^k)$ and the encrypted message $(\ y_1, y_2)$, find the clear text message $(x_1, ..., x_n)$, i.e find integers $x_1, ..., x_n \in [0, 2^s),\ x_{n+k} \in [0, 2^{s+k+\log_2 n+1})$ satisfying

$$\sum_{i=1}^{n+r-1} x_i a_{i,1} = y_1 \bmod q_1 \tag{13}$$

$$\sum_{i=1}^{n+r-1} x_i a_{i,2} = y_2 \tag{14}$$

$$x_{n+k} = \left\lfloor \sum_{i=1}^{n+k-1} x_i f_i^{(k)} \right\rfloor \text{ for } k = 1, ..., r-1 \tag{15}$$

Let's transform these equations into a set of r+1 integer linear equations with m 0-1-unknowns, where $m := ns + (r-1)(r/2 + s + \lceil \log_2 n \rceil - 1) + \sum_{k=1}^{r-1} \operatorname{prec}(k)$.

Since $f_i^{(k)} 2^{\operatorname{prec}(k)} \in [0, 2^{\operatorname{prec}(k)})$ is integral we can write equations 15 as

$$x_{n+k} 2^{\operatorname{prec}(k)} = \sum_{i=1}^{n+k-1} x_i f_i^{(k)} 2^{\operatorname{prec}(k)} - x_{n+r+k-1} \text{ for } k = 1, ..., r-1, \tag{16}$$

where the additional variables $x_{n+r+k-1}$ are integers in $[0, 2^{\operatorname{prec}(k)})$.

With $a_{i,k+2} := \begin{cases} f_i^{(k)} 2^{\operatorname{prec}(k)} & \text{for } i = 1, ..., n+k-1 \\ -2^{\operatorname{prec}(k)} & \text{for } i = n+k \\ 0 & \text{for } i = n+k+1, ..., n+r+k-2 \\ -1 & \text{for } i = n+r+k-1 \\ 0 & \text{for } i = n+r+k, ..., n+2r-2 \end{cases}$

equations 16 simplify to

$$\sum_{i=1}^{n+2r-2} x_i a_{i,k+2} = 0 \text{ for } k = 1, ..., r-1, \tag{17}$$

The unique solution of 13, 14, 17 directly transforms into the unique solution of 13 - 15. To get $\mathtt{0} - \mathtt{1}-$variables we use the binary representation of the integer variables:

We set $d_i := \begin{cases} s & \text{for } 1 \le i \le n \\ s + i + \lceil \log_2 n \rceil - n - 1 & \text{for } n+1 \le i \le n+r-1 \\ \operatorname{prec}(i - (n+r-1)) & \text{for } n+r \le i \le n+2r-2 \end{cases}$

and $D_i := \sum_{j=1}^{i-1} d_j$.

Let $t_{D_i+1}, ..., t_{D_i+d_i} \in \{0, 1\}$ be the binary representation of x_i, i.e.
$x_i = \sum_{l=0}^{d_i-1} t_{D_i+l+1} 2^l$, and set
$A_{D_i+l+1,j} := a_{i,j} 2^l$ for $i = 1, ..., n+2r-2,\ j = 1, ..., r+1,\ l = 0, ..., d_i - 1,$ where $a_{i,1} := a_{i,2} := 0$ for $i > n+r-1$.

With $y_3 := ... := y_{r+1} := 0$ equations 13, 14, 17 simplify to

$$\begin{array}{rcl} \sum_{i=1}^{m} t_i A_{i,1} & = & y_1 + z q_1 \\ \sum_{i=1}^{m} t_i A_{i,j} & = & y_j \text{ for } j = 2, ..., r+1, \\ \text{where} & & t_i \in \{0, 1\},\ z \in \mathbb{Z} \end{array} \tag{18}$$

The row vectors $b_1, ..., b_{m+2} \in \mathbb{Z}^{m+r+2}$ of the following matrix form the basis of lattice

L:

$$\begin{pmatrix} 0 & 2 & 0 & \cdots & 0 & NA_{1,1} & NA_{1,2} & \cdots & NA_{1,r+1} \\ 0 & 0 & 2 & \ddots & 0 & NA_{2,1} & NA_{2,2} & \cdots & NA_{2,r+1} \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots & \\ 0 & 0 & \ddots & 0 & 2 & NA_{2,1} & NA_{2,2} & \cdots & NA_{2,r+1} \\ 0 & 0 & \ddots & 0 & 0 & Nq_1 & 0 & \cdots & 0 \\ 1 & 1 & \cdots & 1 & 1 & Ny_1 & Ny_2 & \cdots & Ny_{r+1} \end{pmatrix} \tag{19}$$

For every integer $N \geq 2$ we can obtain the unique solution $t_1, ..., t_m$ of 18 from each vector $v = (v_0, ..., v_{m+r+1}) = \sum_{i=1}^{m+2} c_i b_i$ with l_∞-norm 1:

v has the form $\{\pm 1\}^{m+1} \times \{0\}^{r+1}$, where $c_{m+2} \in \{\pm 1\}$, $c_{m+1} \in \mathbb{Z}$ and $c_1, ..., c_m \in \{0, -c_{m+2}\}$. The zero in the last $r+1$ coefficients imply

$$\sum_{i=1}^{m} c_i A_{i,1} + c_{m+2} y_1 = 0 \mod q_1 \tag{20}$$

$$\sum_{i=1}^{m} c_i A_{i,j} + c_{m+2} y_j = 0 \text{ for } j = 1, ..., r+1. \tag{21}$$

With $t_i := |c_i| = (|v_i - v_0|)/2$ for $i = 1, ..., m$ we obtain the unique solution of 18 and we directly get the original message from v:

$$x_i := \sum_{j=0}^{s-1} |v_{s(i-1)+j+1} - v_0| 2^{j-1} \text{ for } i = 1, \ldots, n. \tag{22}$$

To find a vector with l_∞-norm 1 we modify algorithm ENUM in order to search for short vectors in l_∞-norm instead of the Euclidean norm $||.||_2$. To do that we make use of **Hoelder's inequality**[22]:

$$|x \cdot y| \leq ||x||_\infty ||y||_1 \text{ forall } x, y \in \mathbb{R}^n. \tag{23}$$

$||y||_1$ is the l_1-norm of y defined as

$$||y||_1 = \sum_{i=1}^{n} |y_i|, \ y \in \mathbb{R}^n. \tag{24}$$

For $t = m, ..., 1$ we define the following functions w_t, $\bar{c}_t$ with integer arguments $\tilde{u}_t, ..., \tilde{u}_m$[23]:

[22]for some background on Hoelders inequality see e.g. `https://en.wikipedia.org/w/index.php?title=Hoelder_inequality`

[23]using the notions of Definition 26

$$w_t := w_t(\tilde{u}_t, ..., \tilde{u}_m) := \pi_t \left(\sum_{i=t}^m \tilde{u}_i b_i\right) = w_{t+1} + \left(\sum_{i=t}^m \tilde{u}_i \mu_{i,t}\right) \hat{b}_t$$
$$\tilde{c}_t := \tilde{c}_t(\tilde{u}_t, ..., \tilde{u}_m) := ||w_t||_2^2 = \tilde{c}_{t+1} + \left(\sum_{i=t}^m \tilde{u}_i \mu_{i,t}\right)^2 ||\hat{b}_t||_2^2$$

Let's have a look into the algorithm ENUM described above (see page 64). It enumerates in depth-first search order all nonzero integer vectors $(\tilde{u}_t, ..., \tilde{u}_m)$ for $t = m, ..., 1$ satisfying $\tilde{c}_t(\tilde{u}_t, ..., \tilde{u}_m) < \bar{c}_1$, where $\bar{c}_1$ is the current minimum for the function $\tilde{c}_1(\tilde{u}_1, ..., \tilde{u}_m)$. In order to find a shortest lattice vector with respect to the l_∞-norm we modify this and recursively enumerate all nonzero integer vectors $(\tilde{u}_t, ..., \tilde{u}_m)$ satisfying $\tilde{c}_t(\tilde{u}_t, ..., \tilde{u}_m) < n\bar{B}^2$, where $\bar{B}$ is the current minimal l_∞-norm of all lattice vectors w_1 enumerated so far. The resulting enumeration area is illustrated in figure 11.1. We enumerate all vectors w_t inside the sphere B with radius $\sqrt{n}\bar{B}$ centered at the origin. We can prune the enumeration using the following observations:
Since, for fixed $\tilde{u}_t, ..., \tilde{u}_m$ we can only reach lattice vectors in the hyperplane H orthogonal to w_t, we can prune the enumeration as soon as this hyperplane doesn't intersect with the set M of all points with l_∞-norm less or equal $\bar{B}$. Using Hoelder's inequality we get $\tilde{c}_t > \bar{B}||w_t||_1$ whenever the intersection is empty. The inequality can be tested in linear time and restricts the enumeration to the shaded area U, i.e. the union of all balls with radius $\frac{1}{2}\sqrt{n}\bar{B}$ centered in $\{\pm\bar{B}/2\}^n$.
The number of vectors w_t to be enumerated and therefore the running time of the enumeration can roughly be approximated by the volume of the area that needs to be traversed. As a consequence the running time of the pruned enumeration algorithm ENUM$_\infty$ below is faster by the factor $volume(U)/volume(B)$. For dimension 2 this factor is exactly $\frac{\pi+2}{2\pi}$ and in dimension n it is approximately $(\frac{\pi+2}{2\pi})^{n-1}$. This means that ENUM$_\infty$ is faster by a factor exponential in the dimension of the lattice. For more details see [13].

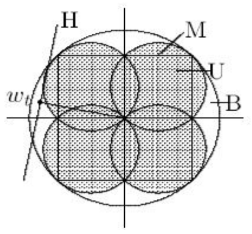


Figure 11.1: Pruning based on Hoelder's inequality (picture created by the author)

We are now able to formulate the attack algorithm:

```
**Algorithm ATTACK-Orton**
Input: public key, encrypted message $y_1, y_2$
1. build the basis $b_1, ..., b_{m+2}$ with $N := n^2$ according to 19
2. LLL-reduce $b_1, ..., b_{m+2}$ with $\delta = 0.99$
3. call ENUM$_\infty$; we get a vector $v$ with $||v||_\infty = 1$
4. $x_i := \sum_{l=0}^{s-1} |v_{s(i-1)+l+1} - v_0| 2^{l-1}$ for $i = 1, ..., n$
Output: original message $x_1, ..., x_n$
```

```
Algorithm ENUM$_\infty$
Input: $\hat{b}_i,\ c_i := ||\hat{b}_i||_2^2,\ \mu_{i,t}$ for $1 \le t \le i \le m$
1. $s := t := 1, \tilde{u}_1 := u_1 := 1, \bar{b} := b_1, \bar{c} := n||b_1||_\infty^2, \bar{B} := ||b_1||_\infty$
   $v_j := y_j := \Delta_j := 0, \delta_j := 1,$
   FOR $i = 1, ..., m+1$
       $\tilde{c}_i := u_i := \tilde{u}_i := v_i := y_i := \Delta_i := 0, \eta_i := \delta_i := 1, w_i := (0, ..., 0)$
2. WHILE $t \le m$
          $\tilde{c}_t := \tilde{c}_{t+1} + (y_t + \tilde{u}_t)^2 c_t$
          IF $\tilde{c}_t < \bar{c}$
          THEN $w_t := w_{t+1} + (y_t + \tilde{u}_t)\hat{b}_t$
               IF $t > 1$
               THEN IF $\tilde{c}_t \ge \bar{B}||w_t||_1$
                    THEN IF $\eta_t = 1$ THEN INCREASE_t()
                         ELSE $\eta_t := 1, \Delta_t := -\Delta_t$
                              IF $\Delta_t \delta_t \ge 0$
                              THEN $\Delta_t := \Delta_t + \delta_t$
                              $\tilde{u}_t := v_t + \Delta_t$
                    ELSE $t := t-1, \eta_t := \Delta_t := 0,$
                        $y_t := \sum_{i=t+1}^{s} \tilde{u}_i \mu_{i,t}, \tilde{u}_t := v_t := \lceil -y_t \rfloor$
                    IF $\tilde{u}_t > -y_t$
                    THEN $\delta_t := -1$
                    ELSE $\delta_t := 1$
               ELSE IF $||w_1||_\infty < \bar{B}$
                    THEN $\bar{b} := w_1, \bar{c} := n||\bar{b}||_\infty^2,$
                         $u_i := \tilde{u}_i$ for $i = 1, ..., m$
          ELSE INCREASE_t()
   END WHILE
Output: $(u_j, ..., u_k), \bar{b}$
```

```
Subroutine INCREASE_t()
$t := t + 1$
$s := \max(t, s)$
IF $\eta_t = 0$
THEN $\Delta_t := -\Delta_t$
        IF $\Delta_t \delta_t \geq 0$ THEN $\Delta_t := \Delta_t + \delta_t$
ELSE $\Delta_t := \Delta_t + \delta_t$
$\tilde{u}_t := v_t + \Delta_t$
```

With the following modifications of ENUM$_\infty$ we can further improve the running time of the attack:
Since $||v||_2^2 = m + 1$ and $||v||_\infty = 1$, we initialize $\bar{c} := m + 1.0001, \bar{B} := 1.0001$ and stop the algorithm as soon as we have found v. We also cut the enumeration for $\tilde{u}_t$ as soon as there is an index $j \in [0, m]$ with $b_{i,j} = 0$ for $i = 1, ..., t - 1$ and $b_{t,j} \neq 0, |w_{t,j}| \neq 1$. We don't miss the solution since $w_{1,j} = w_{t,j} \neq \pm 1$ for all choices of $\tilde{u}_1, ..., \tilde{u}_{t-1}$. As the original basis vectors $b_1, ..., b_{m+1}$ only depend on the public key, we can pre-compute the LLL-reduced basis $b'_1, ..., b'_{m+1}$ of $b_1, ..., b_{m+1}$ once for every public key we want to attack. For all messages which are encrypted with the same public key we use the precomputed vectors $b'_1, ..., b'_{m+1}$ together with b_{m+2} instead of the original basis. More details on the attack including practical results may be found in [12] and [13]

11.2 Factoring

Many public key cryptosystems are based on the assumption that factoring large natural numbers is hard. In 1993 C.P. Schnorr [17] proposed to use lattice basis reduction to factorize natural numbers:

```
Input: $N$ (a natural number with at least two prime factors), $\alpha, c \in \mathbb{Q}$ with $\alpha, c > 1$
1. calculate the list $p_1, ..., p_t$ of the first $t$ primes, $p_t = (\ln N)^\alpha$
2. use lattice basis reduction in order to find $m \geq t+2$ pairs $(u_i, v_i) \in \mathbb{N}^2$ with
       $u_i = \prod_{j=1}^{t} p_j^{a_{i,j}}$ with $a_{i,j} \in \mathbb{N}$        (25)
       $|u_i - v_i N|$ can be factorized over prime factors $p_1, ..., p_t$        (26)
3. factorize $u_i - v_i N$ over primes $p_1, ..., p_t$ and $p_0 = -1$
   Let $u_i - v_i N = \prod_{j=0}^{t} p_j^{b_{i,j}}, b_i = (b_{i,0}, ..., b_{i,t})$ and
   $a_i = (a_{i,0}, ..., a_{i,t})$ with $a_{i,0} = 0$
4. find a 0-1-solution $(c_1, ..., c_m) \neq (0, ..., 0)$ of equation
       $\sum_{i=1}^{m} c_i(a_i + b_i) = 0 \pmod 2$
5. $x := \prod_{j=0}^{t} p_j^{\sum_{i=1}^{m} c_i(a_{i,j}+b_{i,j})/2} \pmod N$
   $y := \prod_{j=0}^{t} p_j^{\sum_{i=1}^{m} c_i b_{i,j}} \pmod N = \prod_{j=0}^{t} p_j^{\sum_{i=1}^{m} c_i a_{i,j}} \pmod N$
   (this construction implies $x^2 = y^2 \pmod N$)
6. if $x \neq y \pmod N$, then output ggT$(x+y, N)$ and stop,
   else goto 4. and find another solution $(c_1, ..., c_m)$
```

In [18] enumeration of short lattice vectors in l_1-norm (similar to ENUM$_\infty$) is used to find the solutions more efficiently. However, those algorithms are still far away from being efficient for large numbers.

11.3 Usage of lattice algorithms in post quantum cryptography and new developments (Eurocrypt 2019)

As it is hard to find the shortest vector in a high-dimensional lattice (in cases when no special "structures" exist, like those found in the Chor-Rivest and Orton cryptosystem), several cryptosystems based on the shortest vector problem are proposed.

The basic idea for constructing lattice-based public-key encryption schemes is to use a "well-formed" high-dimensional lattice basis B as secret key and a scrambled version P of B as public key. For encryption, the sender of a message m maps the message to a point $\overline{m}$ in the lattice, by using the public basis P, and then adds a random error to $\overline{m}$, s.t. the resulting point c is still closer to $\overline{m}$ than to any other point in the lattice. Then, c is sent to the receiver who can use the "well-formed" basis B in order to find $\overline{m}$ efficiently and obtain the original message.

The security of the scheme is based on the assumption, that an attacker who is not in the possession of the "well-formed" basis B, needs to spend an infeasible amount of computational time in order to decipher the message, even with an aid of quantum computers. However, the security of lattice-based schemes against quantum-computer attacks is not yet well-understood. At Eurocrypt 2019 in Darmstadt, several aspects of post quantum cryptography based on lattices were discussed:

A. Pellet-Mary, G. Hanrot, and D. Stehlé [19] describe an algorithm to solve the approximate shortest vector problem for lattices corresponding to ideals of integers of an arbitrary number field K. The running time is still exponential in the input size, but improved compared to previous results.

C. Băetu, F. B. Durak, L. Huguenin-Dumittan, A. Talayhan, and S. Vaudenay [20] describe misuse attacks on several post-quantum cryptosystems proposed by the National Institute of Standards and Technology (NIST) standardization process, including several lattice-based schemes.

M.R. Albrecht, L. Ducas, G. Herold, E. Kirshanova, E.W. Postlethwaite, and M. Stevens [21] propose a sieve method in order to find a shortest lattice vector, or a lattice vector nearest to a given (non-lattice) vector as an alternative to the enumeration algorithms described in this chapter. It would be interesting to check the performance of the enumeration algorithms on modern computers, rather than the implementations on machines as of the late nineties.

12 Appendix: List of the definitions in this chapter

	Short description	Page
Definition 3	vector	12
Definition 4	addition of vectors	13
Definition 5	zero vector	14
Definition 6	vector with dim n	14
Definition 7	linear combination	14
Definition 8	vector norm	14
Definition 9	dot product	15
Definition 10	inner product angel	16
Definition 12	unit vectors	16
Definition 13	sum of matrices	17
Definition 14	scalar multiplies	18
Definition 15	product of matrices	18
Definition 16	transposed matrix	19
Definition 17	minor	21
Definition 18	determinant	22
Definition 19	vector space	26
Definition 20	vector subspace	27
Definition 21	span	27
Definition 22	linear independence	28
Definition 23	vector basis	28
Definition 24	lattice	32
Definition 25	knapsack	34
Definition 26	ordered lattice basis (Gram-Schmidt orthogonalization)	57
Definition 27	i-th-successive minimum	57
Definition 28	HKZ-reduced	58
Definition 29	size-reduced	59
Definition 30	LLL-reduced with δ	60
Definition 31	(β, δ) block-reduced	62

13 Appendix: List of examples in this chapter

	Short description	Page
Example 1	minor	21
Example 2	determinant 2x2	22
Example 3	determinant 3x3	22
Example 4	vector space matrices	27
Example 5	vector space polynomials	27
Example 6	trivial subspace	27
Example 7	polynomial subspace	27
Example 8	geometric subspace	28
Example 9	span	28
Example 10	different basis	28
Example 11	standard basis	29
Example 12	knapsack	36
Example 13	Gauss reduction	58

14 Appendix: List of the challenges in this chapter

	Short description	Page
Challenge 1	hidden message; root polynomials	4
Challenge 2	balanced graph puzzle	6
Challenge 3	system of equations puzzle	10
Challenge 4	coordinates puzzle	15
Challenge 5	the leet challenge	24
Challenge 6	vector space challenge	30
Challenge 7	the superincreasing puzzle	35
Challenge 8	knapsack challenge	38
Challenge 9	LLL algorithm challenge	43
Challenge 10	custom RSA puzzle	47
Challenge 11	Stereotype RSA, e=3, weak pass	55
Challenge 12	Stereotype RSA, e=3, strong pass	55
Challenge 13	Stereotype RSA, e=7, strong pass	56

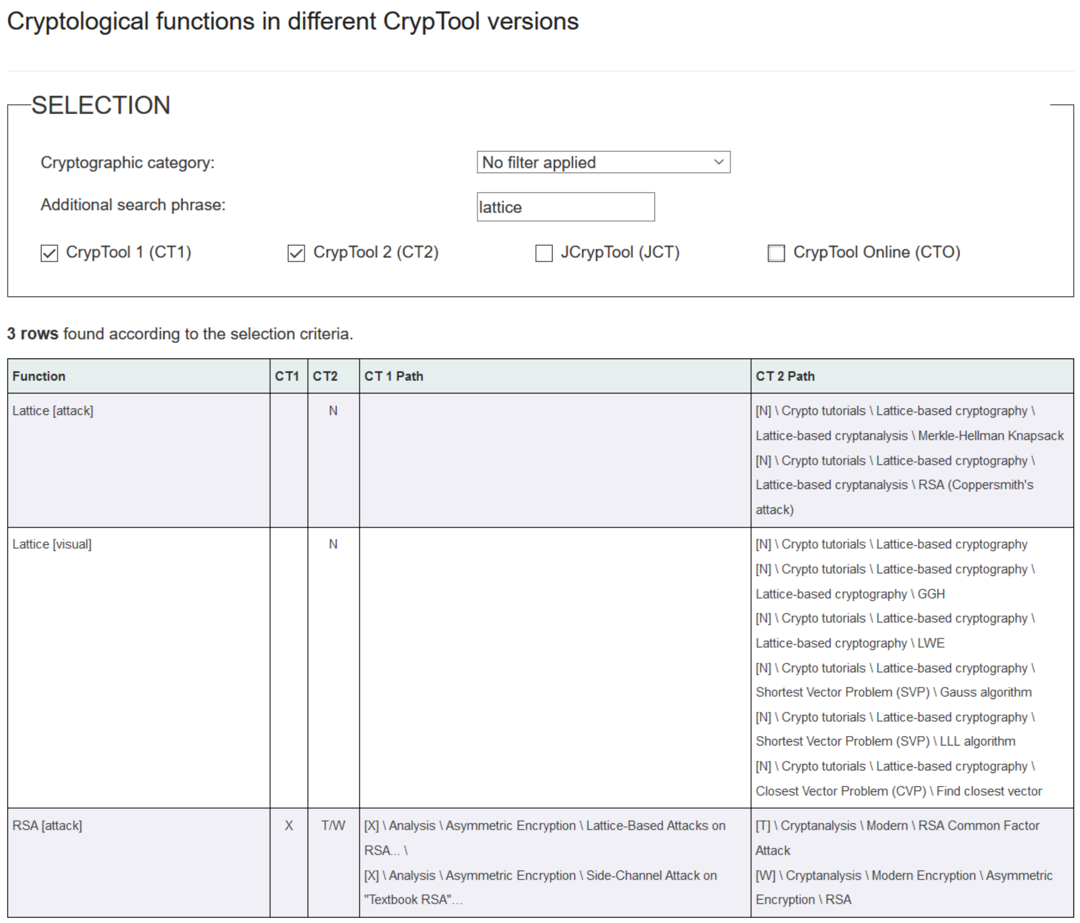
For challenges, where the input is longer, there are text files with the according input. They are bundled in a zip file called `chal_i_helper.zip`. See at the CT website `https://www.cryptool.org/en/ctp-documentation/ctbook`.

15 Appendix: Screenshots and related plugins in the CrypTool programs

The following subsections contain screenshots from CrypTool 1 (CT1), CrypTool 2 (CT2), and JavaCrypTool (JCT) showing plugins dealing with lattices in a didactical manner.

All functions in all CrypTool programs are listed at https://www.cryptool.org/en/ctp-documentation/functionvolume. Specifying a category or a filter string or unboxing programs allows to search for a special function.

Figure 15.1 shows the result, when the selection was restricted to the two programs CT1 and CT2, and the filter string "lattice" was set.



Cryptological functions in different CrypTool versions

SELECTION

Cryptographic category:	No filter applied
Additional search phrase:	lattice

[✓ CrypTool 1 (CT1)] [✓ CrypTool 2 (CT2)] JCrypTool (JCT) CrypTool Online (CTO)

3 rows found according to the selection criteria.

Function	CT1	CT2	CT 1 Path	CT 2 Path
Lattice [attack]		N		[N] \ Crypto tutorials \ Lattice-based cryptography \ Lattice-based cryptanalysis \ Merkle-Hellman Knapsack [N] \ Crypto tutorials \ Lattice-based cryptography \ Lattice-based cryptanalysis \ RSA (Coppersmith's attack)
Lattice [visual]		N		[N] \ Crypto tutorials \ Lattice-based cryptography [N] \ Crypto tutorials \ Lattice-based cryptography \ Lattice-based cryptography \ GGH [N] \ Crypto tutorials \ Lattice-based cryptography \ Lattice-based cryptography \ LWE [N] \ Crypto tutorials \ Lattice-based cryptography \ Shortest Vector Problem (SVP) \ Gauss algorithm [N] \ Crypto tutorials \ Lattice-based cryptography \ Shortest Vector Problem (SVP) \ LLL algorithm [N] \ Crypto tutorials \ Lattice-based cryptography \ Closest Vector Problem (CVP) \ Find closest vector
RSA [attack]	X	T/W	[X] \ Analysis \ Asymmetric Encryption \ Lattice-Based Attacks on RSA... \ [X] \ Analysis \ Asymmetric Encryption \ Side-Channel Attack on "Textbook RSA"...	[T] \ Cryptanalysis \ Modern \ RSA Common Factor Attack [W] \ Cryptanalysis \ Modern Encryption \ Asymmetric Encryption \ RSA

Figure 15.1: Restricted selection from the overview of all CrypTool functions

15.1 Dialogs in CrypTool 1 (CT1)

CT1 contains 4 dialogs dealing with attacks on RSA: The first one is a typical oracle attack (caused by missing padding in plain textbook RSA implementations). The next three use lattices to attack RSA under certain circumstances.[24]

- Textbook RSA
- Factoring with a Hint
- Attack on Stereotyped Messages
- Attack on Small Secret Keys

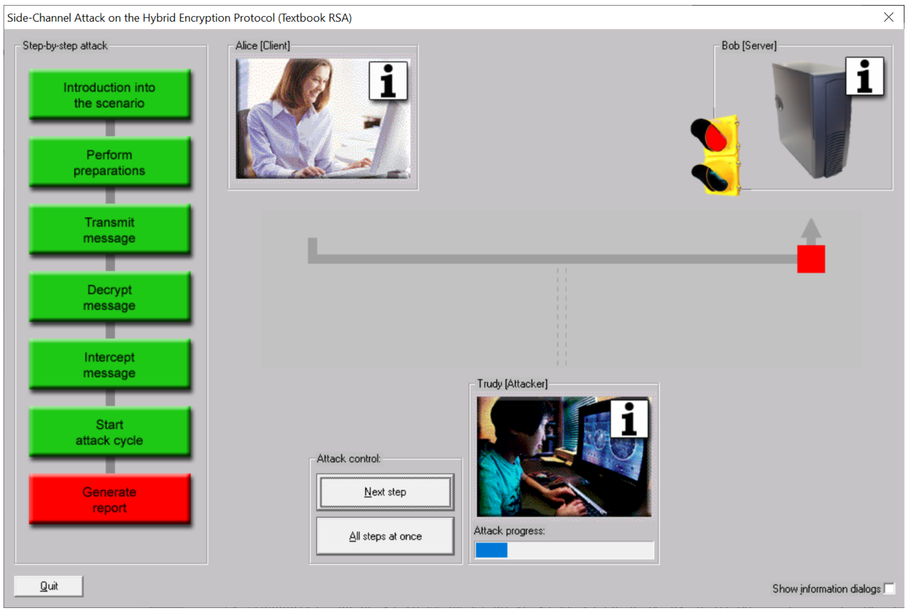
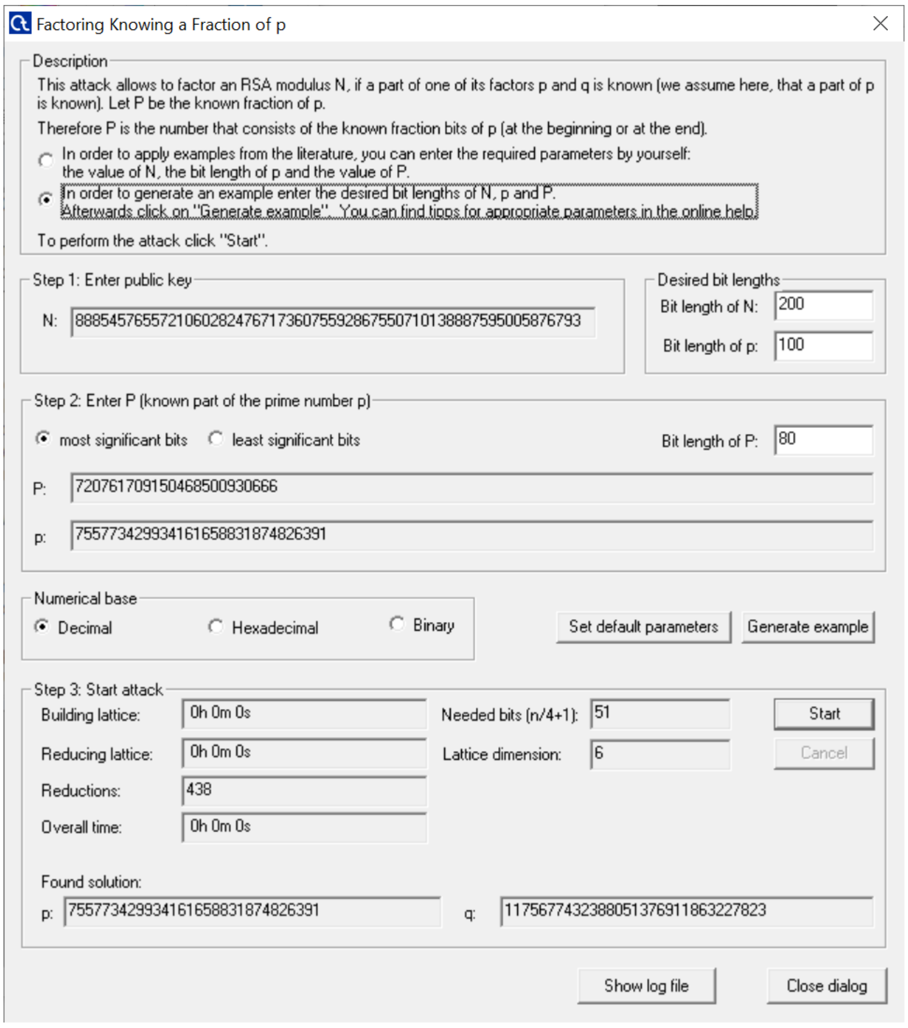


Figure 15.2: CT1 dialog: Side-channel against textbook RSA used in a hybrid encryption protocol (Kuehn 2003)

[24]Within CT1 you can call the first one (Textbook RSA) using the menu path `Analysis->Asymmetric Encryption->Side-Channel Attack on "Textbook RSA"`.
The next three can be found either below the menu `Indiv. Procedures->RSA Cryptosystem->Lattice-Based Attacks` or below `Analysis->Asymmetric Encryption->Lattice-Based Attacks on RSA`.


Factoring Knowing a Fraction of p

Description

This attack allows to factor an RSA modulus N, if a part of one of its factors p and q is known (we assume here, that a part of p is known). Let P be the known fraction of p.

Therefore P is the number that consists of the known fraction bits of p (at the beginning or at the end).

In order to apply examples from the literature, you can enter the required parameters by yourself: the value of N, the bit length of p and the value of P.

[✓ In order to generate an example enter the desired bit lengths of N, p and P. Afterwards click on "Generate example". You can find tipps for appropriate parameters in the online help.]

To perform the attack click "Start".

Step 1: Enter public key

N: 888545765572106028247671736075592867550710138887595005876793

Desired bit lengths

Bit length of N: 200

Bit length of p: 100

Step 2: Enter P (known part of the prime number p)

[✓ most significant bits] least significant bits

Bit length of P: 80

P: 720761709150468500930666

p: 755773429934161658831874826391

Numerical base

[✓ Decimal] Hexadecimal Binary

Step 3: Start attack

Building lattice: 0h 0m 0s

Reducing lattice: 0h 0m 0s

Reductions: 438

Overall time: 0h 0m 0s

Needed bits (n/4+1): 51

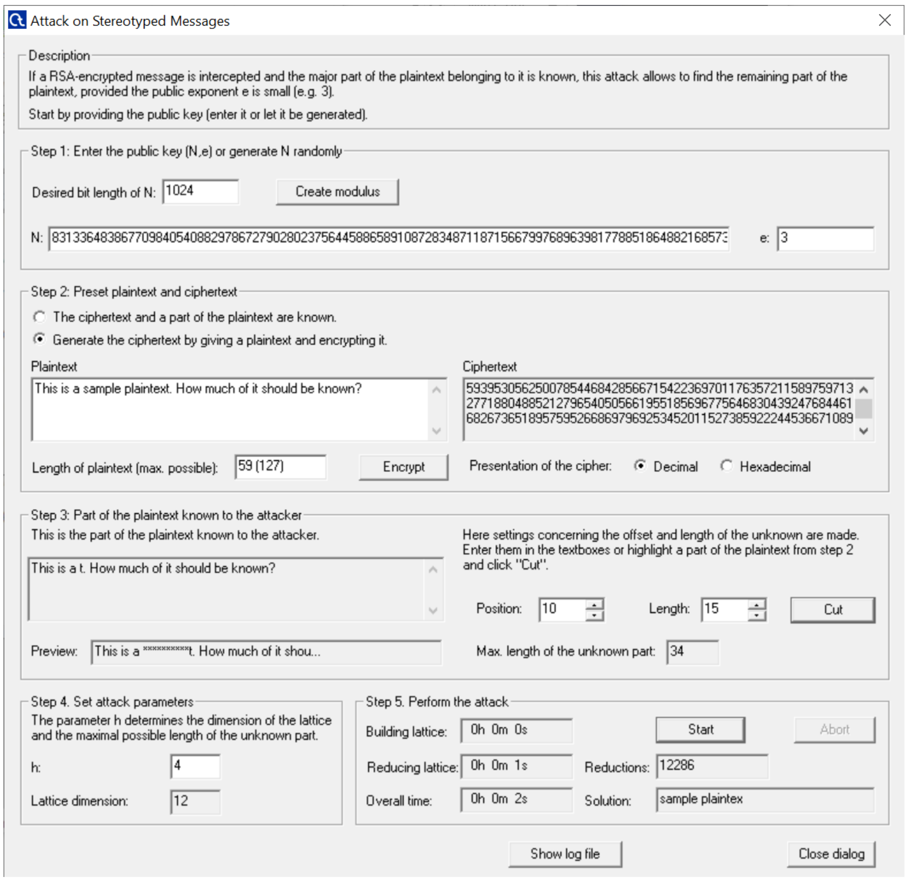
Lattice dimension: 6

Found solution:

p: 755773429934161658831874826391

q: 1175677432388051376911863227823

Figure 15.3: CT1 dialog: Factoring N with a hint (you know a fraction of p)



Attack on Stereotyped Messages

Description

If a RSA-encrypted message is intercepted and the major part of the plaintext belonging to it is known, this attack allows to find the remaining part of the plaintext, provided the public exponent e is small (e.g. 3).

Start by providing the public key (enter it or let it be generated).

Step 1: Enter the public key (N,e) or generate N randomly

Desired bit length of N: 1024

N: 83133648386770984054088297867279028023756445886589108728348711871566799768963981778851864882168573

e: 3

Step 2: Preset plaintext and ciphertext

( ) The ciphertext and a part of the plaintext are known.

[✓ Generate the ciphertext by giving a plaintext and encrypting it.]

Plaintext: This is a sample plaintext. How much of it should be known?

Ciphertext:

```
59395305625007854468428566715422369701176357211589759713
27718804885212796540505661955185696775646830439247684461
68267365189575952668697969253452011527385922244536671089
```

Length of plaintext (max. possible): 59 (127)

Presentation of the cipher: [✓ Decimal] Hexadecimal

Step 3: Part of the plaintext known to the attacker

This is the part of the plaintext known to the attacker.

This is a t. How much of it should be known?

Preview: This is a **********t. How much of it shou...

Here settings concerning the offset and length of the unknown are made. Enter them in the textboxes or highlight a part of the plaintext from step 2 and click "Cut".

Position: 10 Length: 15

Max. length of the unknown part: 34

Step 4. Set attack parameters

The parameter h determines the dimension of the lattice and the maximal possible length of the unknown part.

h: 4

Lattice dimension: 12

Step 5. Perform the attack

Building lattice: 0h 0m 0s

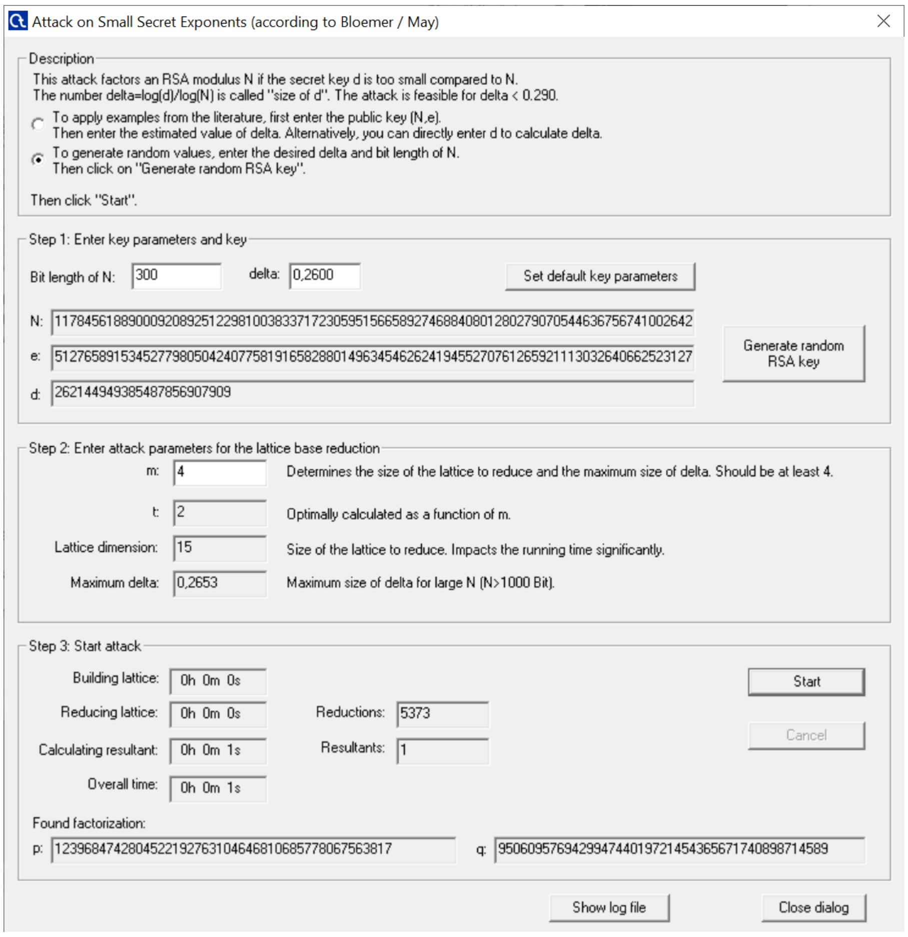
Reducing lattice: 0h 0m 1s

Reductions: 12286

Overall time: 0h 0m 2s

Solution: sample plaintex

Figure 15.4: CT1 dialog: Attack on stereotyped messages (you know a part of the plaintext message)



Attack on Small Secret Exponents (according to Bloemer / May)

Description

This attack factors an RSA modulus N if the secret key d is too small compared to N.
The number delta=log(d)/log(N) is called "size of d". The attack is feasible for delta < 0.290.

( ) To apply examples from the literature, first enter the public key (N,e).
Then enter the estimated value of delta. Alternatively, you can directly enter d to calculate delta.

(●) To generate random values, enter the desired delta and bit length of N.
Then click on "Generate random RSA key".

Then click "Start".

Step 1: Enter key parameters and key

Bit length of N: 300 delta: 0,2600

N: 117845618890009208925122981003833717230595156658927468840801280279070544636756741002642

e: 512765891534527798050424077581916582880149634546262419455270761265921113032640662523127

d: 262144949385487856907909

Step 2: Enter attack parameters for the lattice base reduction

m: 4 Determines the size of the lattice to reduce and the maximum size of delta. Should be at least 4.

t: 2 Optimally calculated as a function of m.

Lattice dimension: 15 Size of the lattice to reduce. Impacts the running time significantly.

Maximum delta: 0,2653 Maximum size of delta for large N (N>1000 Bit).

Step 3: Start attack

Building lattice: 0h 0m 0s

Reducing lattice: 0h 0m 0s Reductions: 5373

Calculating resultant: 0h 0m 1s Resultants: 1

Overall time: 0h 0m 1s

Found factorization:

p: 1239684742804522192763104646810685778067563817 q: 950609576942994744019721454365671740898714589

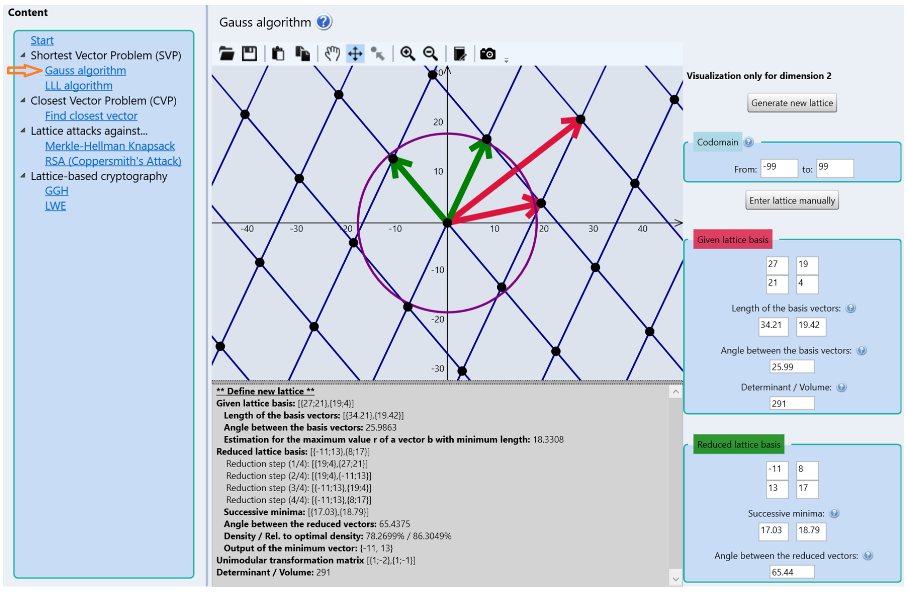
Figure 15.5: CT1 dialog: Factoring N when the private exponent is too small (Bloemer/May 2001)

15.2 Lattice tutorial in CrypTool 2 (CT2)

CT2 contains a plugin “Lattice-based cryptography” with the following playful introductory programs.[25]

- Algorithms to reduce lattice basis for shortest vector problem (SVP):
  - Gauss (nice visualization in 2-dim)
  - LLL
- Closest vector problem (CVP)
  - Find closest vector (nice visualization in 2-dim)
- Lattice-based attacks against:
  - Merkle-Hellman knapsack
  - RSA (Coppersmith attack)
- Lattice-based cryptography:
  - GGH
  - LWE

[25] CT2 contains visualizations of these methods within the crypto tutorial **Lattice-based cryptography** below the main menu **Crypto tutorials**. Please note, that most of the plugins in CT2 appear in the workspace manager as templates and components to be started from the **Startcenter** (and not via the menu **Crypto tutorials**).


Content
Start
Shortest Vector Problem (SVP)
Gauss algorithm
LLL algorithm
Closest Vector Problem (CVP)
Find closest vector
Lattice attacks against...
Merkle-Hellman Knapsack
RSA (Coppersmith's Attack)
Lattice-based cryptography
GGH
LWE
Gauss algorithm
Visualization only for dimension 2
Generate new lattice
Enter lattice manually

Codomain
From: -99 to: 99

Given lattice basis

27	19
21	4

Length of the basis vectors: 34.21 19.42
Angle between the basis vectors: 25.99
Determinant / Volume: 291

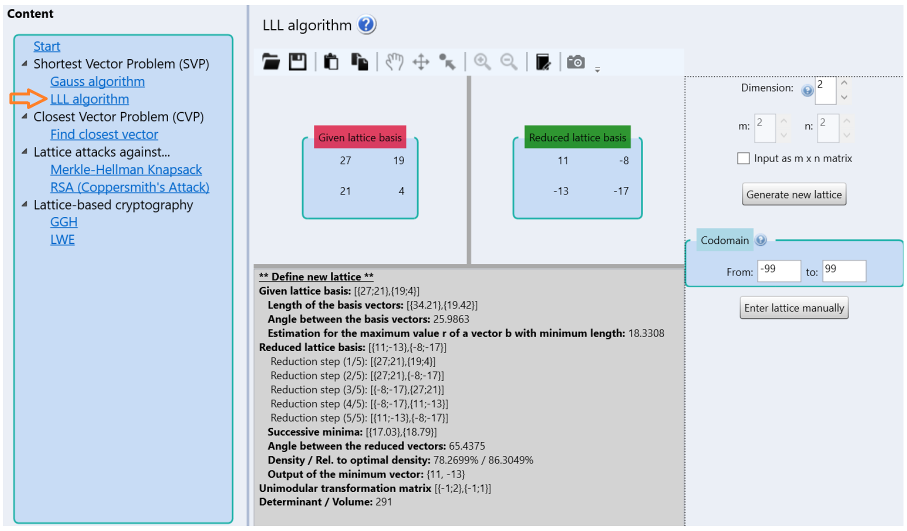
Reduced lattice basis

-11	8
13	17

Successive minima: 17.03 18.79
Angle between the reduced vectors: 65.44

```
** Define new lattice **
Given lattice basis: [{27;21},{19;4}]
  Length of the basis vectors: [{34.21},{19.42}]
  Angle between the basis vectors: 25.9863
  Estimation for the maximum value r of a vector b with minimum length: 18.3308
Reduced lattice basis: [{-11;13},{8;17}]
   Reduction step (1/4): [{19;4},{27;21}]
   Reduction step (2/4): [{19;4},{-11;13}]
   Reduction step (3/4): [{-11;13},{19;4}]
   Reduction step (4/4): [{-11;13},{8;17}]
  Successive minima: [{17.03},{18.79}]
  Angle between the reduced vectors: 65.4375
  Density / Rel. to optimal density: 78.2699% / 86.3049%
  Output of the minimum vector: {-11, 13}
Unimodular transformation matrix [{1;-2},{1;-1}]
Determinant / Volume: 291
```

Figure 15.6: CT2 tutorial "Lattice-based cryptography": SVP via Gauss



Given lattice basis

27	19
21	4

Reduced lattice basis

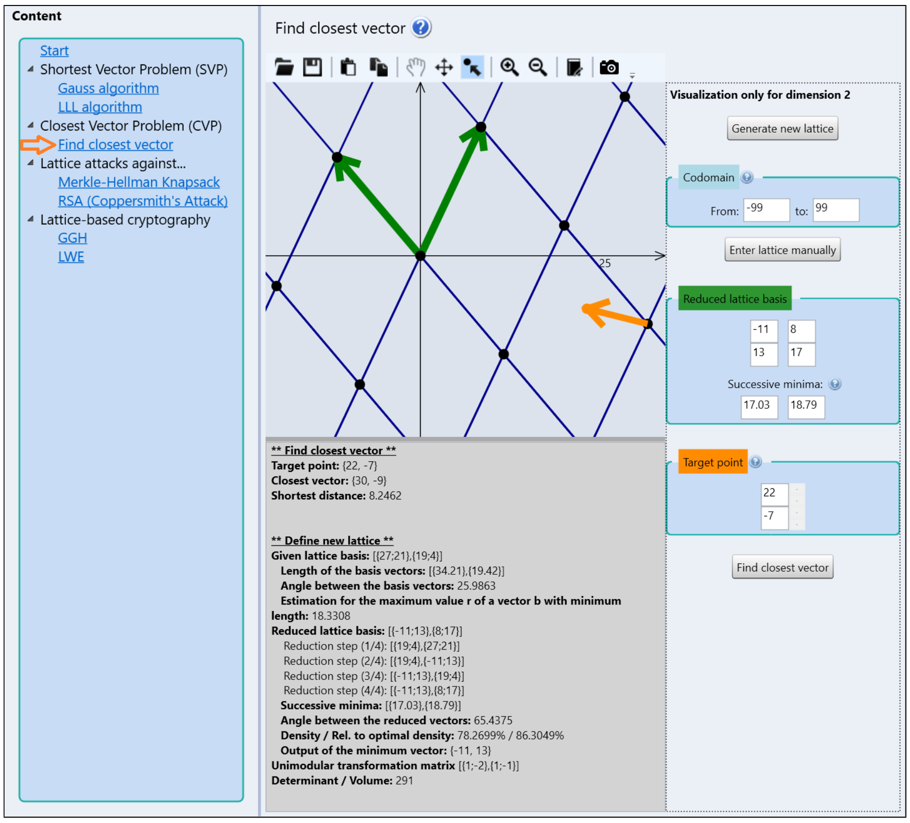
11	-8
-13	-17

Dimension: 2
m: 2 n: 2
☐ Input as m x n matrix

Codomain
From: -99 to: 99

```
** Define new lattice **
Given lattice basis: [{27;21},{19;4}]
  Length of the basis vectors: [{34.21},{19.42}]
  Angle between the basis vectors: 25.9863
  Estimation for the maximum value r of a vector b with minimum length: 18.3308
Reduced lattice basis: [{11;-13},{-8;-17}]
   Reduction step (1/5): [{27;21},{19;4}]
   Reduction step (2/5): [{27;21},{-8;-17}]
   Reduction step (3/5): [{-8;-17},{27;21}]
   Reduction step (4/5): [{-8;-17},{11;-13}]
   Reduction step (5/5): [{11;-13},{-8;-17}]
  Successive minima: [{17.03},{18.79}]
  Angle between the reduced vectors: 65.4375
  Density / Rel. to optimal density: 78.2699% / 86.3049%
  Output of the minimum vector: {11, -13}
Unimodular transformation matrix [{-1;2},{-1;1}]
Determinant / Volume: 291
```

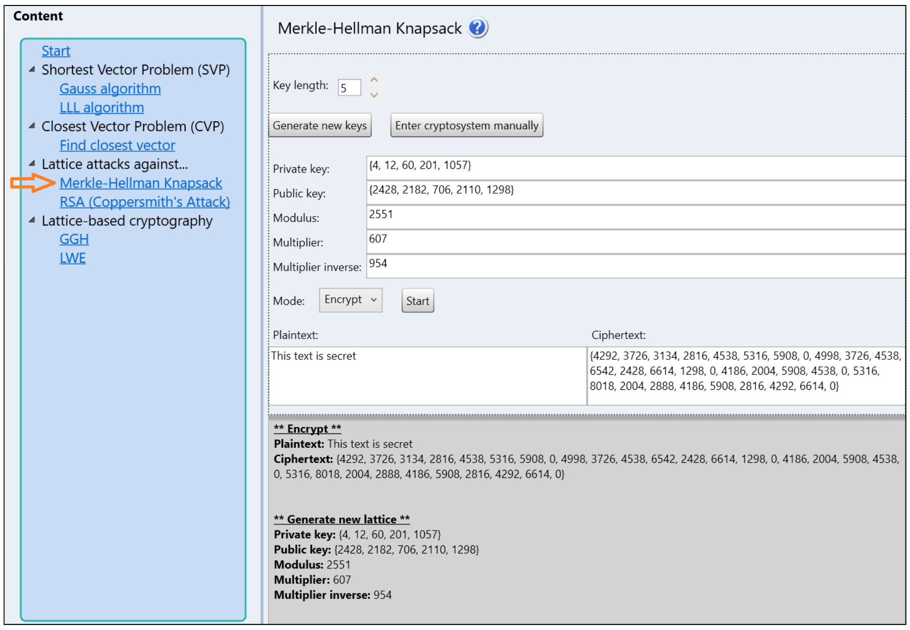
Figure 15.7: CT2 tutorial "Lattice-based cryptography": SVP via LLL algorithm



** Find closest vector **
Target point: {22, -7}
Closest vector: {30, -9}
Shortest distance: 8.2462

** Define new lattice **
Given lattice basis: [{27;21},{19;4}]
Length of the basis vectors: [{34.21},{19.42}]
Angle between the basis vectors: 25.9863
Estimation for the maximum value r of a vector b with minimum length: 18.3308
Reduced lattice basis: [{-11;13},{8;17}]
Reduction step (1/4): [{19;4},{27;21}]
Reduction step (2/4): [{19;4},{-11;13}]
Reduction step (3/4): [{-11;13},{19;4}]
Reduction step (4/4): [{-11;13},{8;17}]
Successive minima: [{17.03},{18.79}]
Angle between the reduced vectors: 65.4375
Density / Rel. to optimal density: 78.2699% / 86.3049%
Output of the minimum vector: {-11, 13}
Unimodular transformation matrix [{1;-2},{1;-1}]
Determinant / Volume: 291

Figure 15.8: CT2 tutorial “Lattice-based cryptography”: CVP – Find closest vector


Content
Start
Shortest Vector Problem (SVP)
Gauss algorithm
LLL algorithm
Closest Vector Problem (CVP)
Find closest vector
Lattice attacks against...
Merkle-Hellman Knapsack
RSA (Coppersmith's Attack)
Lattice-based cryptography
GGH
LWE
Merkle-Hellman Knapsack
Generate new keys
Enter cryptosystem manually
Start

Key length: 5

Private key: {4, 12, 60, 201, 1057}
Public key: {2428, 2182, 706, 2110, 1298}
Modulus: 2551
Multiplier: 607
Multiplier inverse: 954

Mode: Encrypt

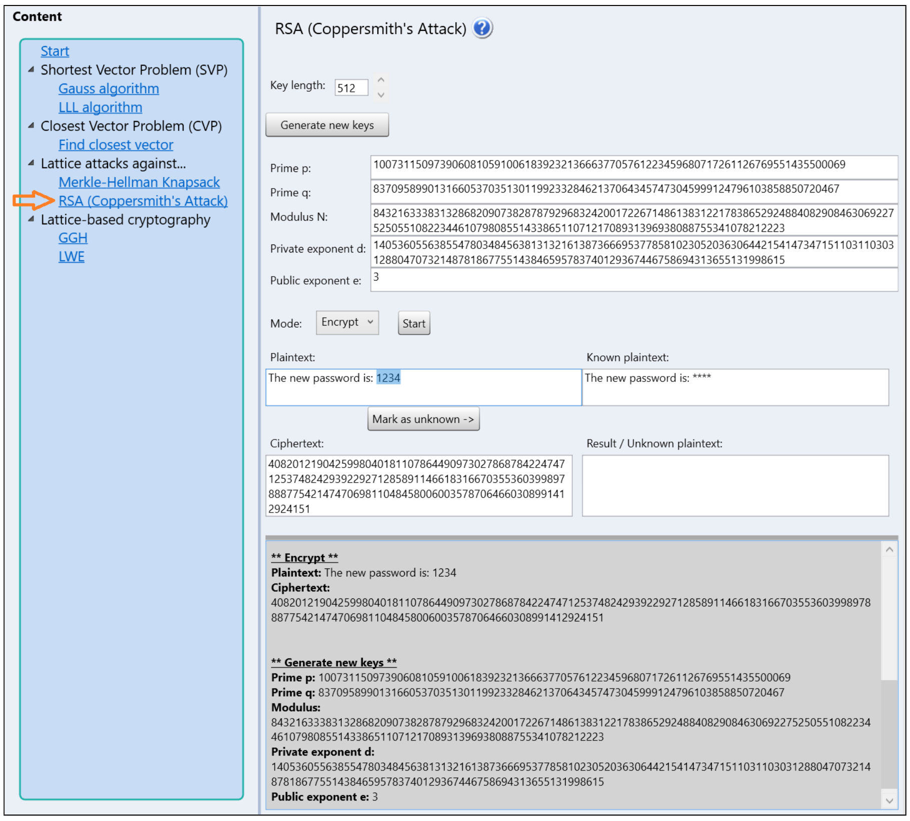
Plaintext:
This text is secret

Ciphertext:
{4292, 3726, 3134, 2816, 4538, 5316, 5908, 0, 4998, 3726, 4538, 6542, 2428, 6614, 1298, 0, 4186, 2004, 5908, 4538, 0, 5316, 8018, 2004, 2888, 4186, 5908, 2816, 4292, 6614, 0}

<u> Encrypt **</u>**
Plaintext: This text is secret
Ciphertext: {4292, 3726, 3134, 2816, 4538, 5316, 5908, 0, 4998, 3726, 4538, 6542, 2428, 6614, 1298, 0, 4186, 2004, 5908, 4538, 0, 5316, 8018, 2004, 2888, 4186, 5908, 2816, 4292, 6614, 0}

**** Generate new lattice ****
Private key: {4, 12, 60, 201, 1057}
Public key: {2428, 2182, 706, 2110, 1298}
Modulus: 2551
Multiplier: 607
Multiplier inverse: 954

Figure 15.9: CT2 tutorial "Lattice-based cryptography": attack against the Merkle-Hellman knapsack cryptosystem


Content
Start
Shortest Vector Problem (SVP)
Gauss algorithm
LLL algorithm
Closest Vector Problem (CVP)
Find closest vector
Lattice attacks against...
Merkle-Hellman Knapsack
RSA (Coppersmith's Attack)
Lattice-based cryptography
GGH
LWE
RSA (Coppersmith's Attack)
Generate new keys
Start
Mark as unknown ->

Key length: 512

Prime p: 10073115097390608105910061839232136663770576122345968071726112676955143550006 9

Prime q: 83709589901316605370351301199233284621370643457473045999124796103858850720467

Modulus N: 8432163338313286820907382878792968324200172267148613831221783865292488408290846306922752505510822344610798085514338651107121708931396938088755341078212223

Private exponent d: 14053605563855478034845638131321613873666953778581023052036306442154147347151103110303128804707321487818677551438465957837401293674467586943136551319986 15

Public exponent e: 3

Mode: Encrypt

Plaintext: The new password is: 1234

Known plaintext: The new password is: ****

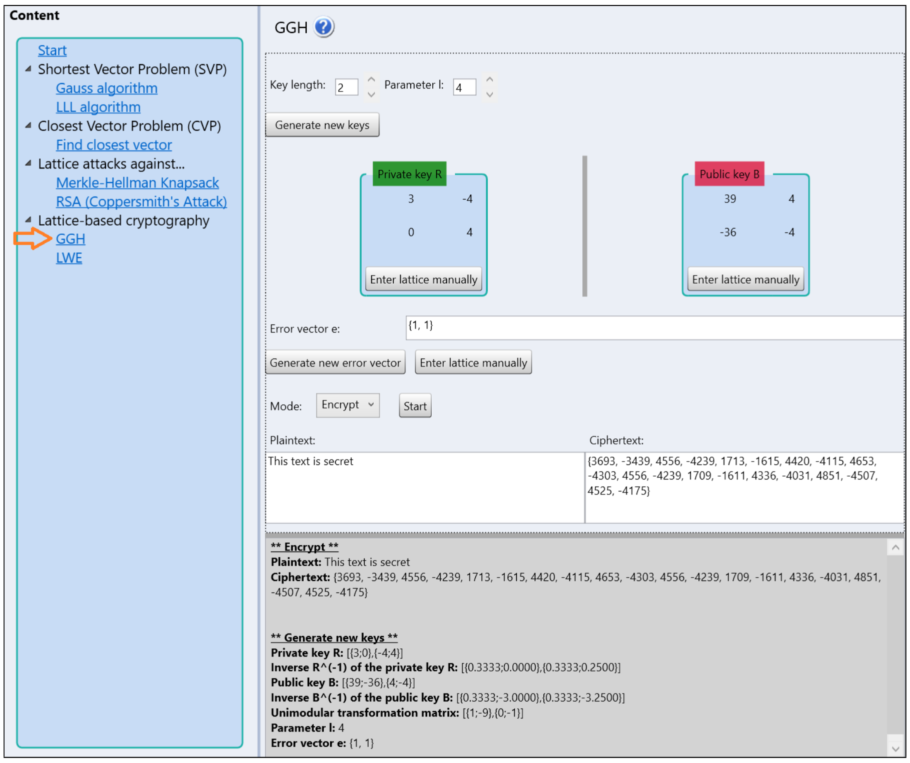
Ciphertext:
```
408201219042599804018110786449097302786878422474 7
125374824293922927128589114661831667035536039989 7
888775421474706981104845800600357870646603089914 1
2924151
```

Result / Unknown plaintext:

```
** Encrypt **
Plaintext: The new password is: 1234
Ciphertext:
40820121904259980401811078644909730278687842247471253748242939229271285891146618316670355360399897 8
88775421474706981104845800600357870646603089914 12924151


** Generate new keys **
Prime p: 10073115097390608105910061839232136663770576122345968071726112676955143550006 9
Prime q: 83709589901316605370351301199233284621370643457473045999124796103858850720467
Modulus:
84321633383132868209073828787929683242001722671486138312217838652924884082908463069227525055108223 4
4610798085514338651107121708931396938088755341078212223
Private exponent d:
14053605563855478034845638131321613873666953778581023052036306442154147347151103110303128804707321 4
8781867755143846595783740129367446758694313655131998615
Public exponent e: 3
```

Figure 15.10: CT2 tutorial “Lattice-based cryptography”: attack against RSA (Coppersmith)



Key length: 2 Parameter l: 4

Private key R	
3	-4
0	4

Public key B	
39	4
-36	-4

Error vector e: {1, 1}

Mode: Encrypt

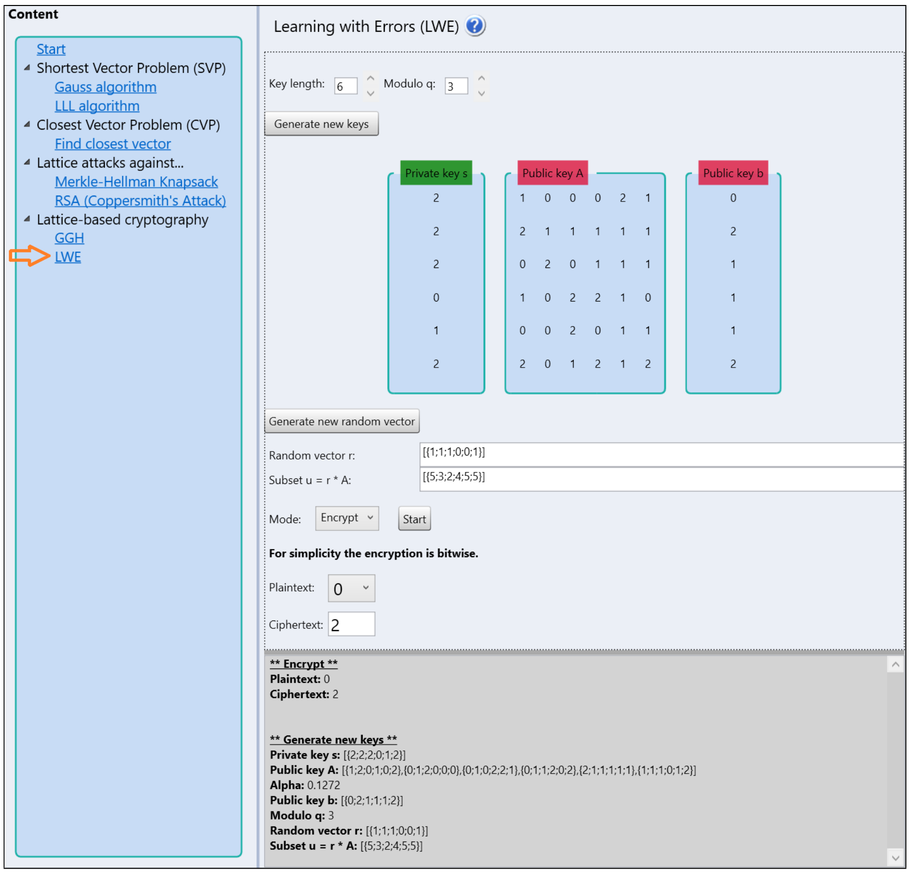
Plaintext: This text is secret

Ciphertext: {3693, -3439, 4556, -4239, 1713, -1615, 4420, -4115, 4653, -4303, 4556, -4239, 1709, -1611, 4336, -4031, 4851, -4507, 4525, -4175}

```
** Encrypt **
Plaintext: This text is secret
Ciphertext: {3693, -3439, 4556, -4239, 1713, -1615, 4420, -4115, 4653, -4303, 4556, -4239, 1709, -1611, 4336, -4031, 4851,
-4507, 4525, -4175}


** Generate new keys **
Private key R: [{3;0},{-4;4}]
Inverse R^(-1) of the private key R: [{0.3333;0.0000},{0.3333;0.2500}]
Public key B: [{39;-36},{4;-4}]
Inverse B^(-1) of the public key B: [{0.3333;-3.0000},{0.3333;-3.2500}]
Unimodular transformation matrix: [{1;-9},{0;-1}]
Parameter l: 4
Error vector e: {1, 1}
```

Figure 15.11: CT2 tutorial “Lattice-based cryptography”: the GGH cryptosystem


Content
Start
Shortest Vector Problem (SVP)
Gauss algorithm
LLL algorithm
Closest Vector Problem (CVP)
Find closest vector
Lattice attacks against...
Merkle-Hellman Knapsack
RSA (Coppersmith's Attack)
Lattice-based cryptography
GGH
LWE
Learning with Errors (LWE)
Generate new keys
Generate new random vector
Start

Key length: 6 Modulo q: 3

Private key s	Public key A						Public key b
2	1	0	0	0	2	1	0
2	2	1	1	1	1	1	2
2	0	2	0	1	1	1	1
0	1	0	2	2	1	0	1
1	0	0	2	0	1	1	1
2	2	0	1	2	1	2	2

Random vector r: [{1;1;1;0;0;1}]
Subset u = r * A: [{5;3;2;4;5;5}]

Mode: Encrypt

For simplicity the encryption is bitwise.

Plaintext: 0
Ciphertext: 2

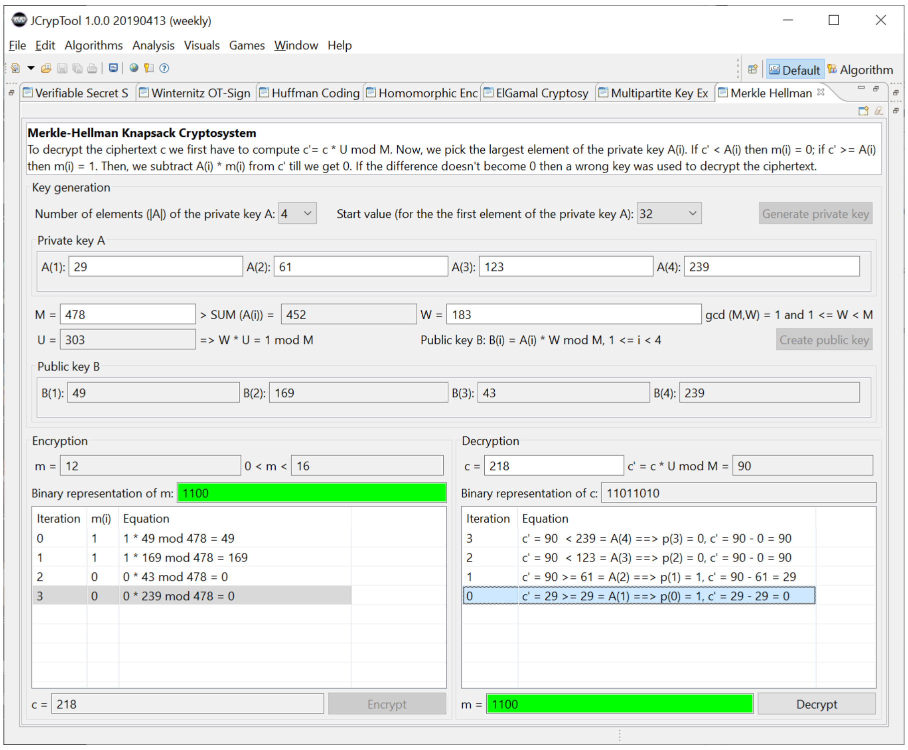
```
** Encrypt **
Plaintext: 0
Ciphertext: 2


** Generate new keys **
Private key s: [{2;2;2;0;1;2}]
Public key A: [{1;2;0;1;0;2},{0;1;2;0;0;0},{0;1;0;2;2;1},{0;1;1;2;0;2},{2;1;1;1;1;1},{1;1;1;0;1;2}]
Alpha: 0.1272
Public key b: [{0;2;1;1;1;2}]
Modulo q: 3
Random vector r: [{1;1;1;0;0;1}]
Subset u = r * A: [{5;3;2;4;5;5}]
```

Figure 15.12: CT2 tutorial “Lattice-based cryptography”: the LWE cryptosystem

15.3 Plugin in JCrypTool (JCT)

JCT contains a visualization of the Merkle-Hellman knapsack cryptosystem.[26] This is just a didactical visualization showing all the necessary steps for private keys with maximum 20 elements.



Merkle-Hellman Knapsack Cryptosystem
To decrypt the ciphertext c we first have to compute c'= c * U mod M. Now, we pick the largest element of the private key A(i). If c' < A(i) then m(i) = 0; if c' >= A(i) then m(i) = 1. Then, we subtract A(i) * m(i) from c' till we get 0. If the difference doesn't become 0 then a wrong key was used to decrypt the ciphertext.

Key generation

Number of elements (|A|) of the private key A: 4
Start value (for the the first element of the private key A): 32
Generate private key

Private key A

A(1): 29 A(2): 61 A(3): 123 A(4): 239

M = 478 > SUM (A(i)) = 452 W = 183 gcd (M,W) = 1 and 1 <= W < M
U = 303 => W * U = 1 mod M Public key B: B(i) = A(i) * W mod M, 1 <= i < 4
Create public key

Public key B

B(1): 49 B(2): 169 B(3): 43 B(4): 239

Encryption

m = 12 0 < m < 16

Binary representation of m: 1100

Iteration	m(i)	Equation
0	1	1 * 49 mod 478 = 49
1	1	1 * 169 mod 478 = 169
2	0	0 * 43 mod 478 = 0
3	0	0 * 239 mod 478 = 0

c = 218 Encrypt

Decryption

c = 218 c' = c * U mod M = 90

Binary representation of c: 11011010

Iteration	Equation
3	c' = 90 < 239 = A(4) ==> p(3) = 0, c' = 90 - 0 = 90
2	c' = 90 < 123 = A(3) ==> p(2) = 0, c' = 90 - 0 = 90
1	c' = 90 >= 61 = A(2) ==> p(1) = 1, c' = 90 - 61 = 29
0	c' = 29 >= 29 = A(1) ==> p(0) = 1, c' = 29 - 29 = 0

m = 1100 Decrypt

Figure 15.13: JCT plugin: Merkle-Hellman knapsack cryptosystem: step-by-step calculations

[26]JCT offers this within the Default Perspective in the menu **Visuals \ Merkle-Hellman Knapsack Cryptosystem**.

16 Authors of this Chapter

This appendix lists the authors of this document.
Please refer to the top of each individual chapter for their contribution.

Miroslav Dimitrov
PhD student at the Bulgarian Academy of Sciences
E-mail: mirdim@math.bas.bg

Harald Ritter
Member of IACR, PhD thesis on lattice basis reduction at the University of Frankfurt.
Senior Consultant at NOVOSEC AG, Frankfurt/Main.
E-mail: harald.ritter@novosec.com, harald.ritter@onlinehome.de

Bernhard Esslinger
Initiator of the CrypTool project, editor and main author of this book. Professor for IT security and cryptography at the University of Siegen.
E-mail: bernhard.esslinger@gmail.com, bernhard.esslinger@uni-siegen.de

The authors want to thank Dr. Doris Behrendt for proof reading and correcting this chapter.

Bibliography

[1] Lenstra, Arjen Klaas, Hendrik Willem Lenstra, and László Lovász: Factoring polynomials with rational coefficients. Mathematische Annalen 261.4 (1982): 515-534.

[2] May, Alexander: Using LLL-Reduction for Solving RSA and Factorization Problems. The LLL algorithm. Springer, Berlin, Heidelberg, 2009. 315-348.

[3] NIST: FIPS PUB 186-4: Digital Signature Standard (DSS), 2013, https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.186-4.pdf

[4] Coppersmith, Don: Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. Journal of Cryptology 10.4 (1997): 233-260.

[5] Howgrave-Graham, Nicholas A.: Computational Mathematics Inspired by RSA. Dissertation, University of Bath, 1998, https://cr.yp.to/bib/1998/howgrave-graham.pdf

[6] Sang-Gu Lee: Linear Algebra with Sage, 2018, https://www.researchgate.net/publication/327362474_Linear_Algebra_seonhyeongdaesuhag_e-_book_-_2018_version.

[7] Hermite, C.: Extraits de lettres de M. Ch. Hermite à M. Jacobi sur differents objets de de la théorie des nombres, deuxième lettre. Journal reine angewandte Mathematik, 40:279-290, 1850.

[8] Korkine, A. and G. Zolotarev: Sur les formes quadratiques positives quaternaires. Math. Annalen, 5:581-583, 1872.

[9] Korkine, A. and G. Zolotarev: Sur les formes quadratiques. Math. Annalen, 6:366-389, 1873.

[10] Korkine, A. and G. Zolotarev: Sur les formes quadratiques positives. Math. Annalen, 11:242-292, 1877.

[11] Schnorr, C.P. and M. Euchner: Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. Mathematical Programming (1994), 66:181-199.

[12] Ritter, H.: Breaking knapsack cryptosystems by l_∞-norm enumeration. In J. Pribyl, editor, Proceedings of the 1st International Conference on the Theory and Applications of Cryptology, PRAGOCRYPT '96, Prague, Czech Republic, 30 September-3 October, 1996, CTU Publishing House, pp. 480-492, 1996.

[13] Ritter, H.: Aufzählung von kurzen Gittervektoren in allgemeiner Norm. Dissertation zur Erlangung des Doktorgrades der Naturwissenschaften, Fachbereich Mathematik der Johann Wolfgang Goethe-Universität in Frankfurt am Main, 1997.

[14] Chor, B and R.L. Rivest: A Knapsack Type Public-Key Cryptosystem Based on Arithmetic in Finite Fields. IEEE Transactions on Information Theory, 34(5):901-999, September 1988.

[15] Orton, G.: A Multiple-Iterated Trapdoor for Dense Compact Knapsacks. In Proc. EUROCRYPT 94, pp. 112-130. Springer Lecture Notes in Computer Science, No. 950, 1994.

[16] Schnorr, C.P. and H.H. Hörner: Attacking the Chor-Rivest Cryptosystem by Improved Lattice Reduction. In Proc. EUROCRYPT 95, pp. 1-12. Springer Lecture Notes in Computer Science, No. 921, 1995.

[17] Schnorr, C.P.: Factoring Integers and Computing Discrete Logarithms via Diophantine Approximations. In Advances of Computational Complexity, DIMACS Series in Discrete Mathematics and Theoretical Science, AMS; 1993.

[18] Ritter, H. and C. Rössner: Factoring via Strong Lattice Reduction Algorithms, Technical Report, available at https://www.researchgate.net/publication/2266562_Factoring_via_Strong_Lattice_Reduction_Algorithms,1997.

[19] Pellet-Mary, A., G. Hanrot, and D. Stehlé: Approx-SVP in Ideal Lattices with Preprocessing. In Proc. EUROCRYPT 19, pp. 685-716. Springer Lecture Notes in Computer Science, 2019. Also available at https://eprint.iacr.org/2019/215.pdf.

[20] Băetu, C., F.B. Durak, L. Huguenin-Dumittan, A. Talayhan, and S. Vaudenay: Misuse Attacks on Post-Quantum Cryptosystems. In Proc. EUROCRYPT 19, pp. 747-776. Springer Lecture Notes in Computer Science, 2019. Also available at https://eprint.iacr.org/2019/525.

[21] Albrecht, M.R., L. Ducas, G. Herold, E. Kirshanova, E.W. Postlethwaite, and M. Stevens: The General Sieve Kernel and New Records in Lattice Reduction. In Proc. EUROCRYPT 19, pp. 717-746. Springer Lecture Notes in Computer Science, 2019. Also available at https://eprint.iacr.org/2019/089.