

Vorlesung Nr. 4

Kryptologie II - Datum: 11.10.2018

- Modulare Arithmetik
- Mathematische Grundlagen

Buch der Woche

Titel: Cracking Codes with Python

Autor: Al Sweigart

Verlag: no starch press

Umfang: ca. 380 Seiten

Hinweise zum Inhalt:

- Einführung in Python (startet bei Null) anhand kryptographischer Fragestellungen
- betrachtet klassische Chiffren, Sprachanalyse mittels Statistik, Primzahlen, Public Key Crypto; Schwerpunkt liegt auf historischen Verfahren, zwecks Einführung der jeweiligen Programmierkonstrukte
- sehr viele einfache Code-Beispiele, gut verständlich
- Webseite mit Begleitmaterial im Internet
- Empfehlung für alle, die in Python noch nicht zu fortgeschritten sind

Operators in Python

The floor Operator

For some operations like integer division, for example, Python needs to compute the floor of a number. In Mathematics, we denote the floor of x by $\lfloor x \rfloor$.

The floor of a number x is the nearest integer below or equal to it:

For $x \in \mathbb{R}$: $\lfloor x \rfloor = \max \{k \in \mathbb{Z} : k \leq x\}$.

Out of the box, Python doesn't provide a function called floor. We need to include the math module. Then we can use `math.floor()` as follows:

Example:

```
>>> math.floor(5.2)
```

```
5
```

```
>>> math.floor(-5.2)
```

```
-6
```

The Integer Division Operator //

Integer Division uses the operator //

Python doesn't round the result of an integer division. Instead, the result is the floor of the floating-point division result.

Example:

$5 // 3$ is by definition equal to $\lfloor \frac{5}{3} \rfloor$ which returns 1.

```
>>>> 5 // 3
```

```
1 ←
```

```
>>>> -5 // 3
```

```
-2 ←
```

Remark: the (non-integer) division operator / always delivers a floating-point result, even if both operands are integers. This holds for Python 3 and was different in Python 2.

Example:

```
>>> 6 / 3
```

```
2.0 ←
```

The // Operator on Floating-Points (1)


The operator `//` also allows floating-point numbers as operands. Again, the result is the nearest integer below or equal to the floating-point division result.

Example:

```
>>>> 5.2 // 3.1  
1.0
```

Note that the result now is a floating-point number, too – and not an integer.

As we know, floating-point arithmetic works with approximations and produces rounding errors. For example, we don't necessarily get an exact binary representation of the rational number 0.1. This leads to the following problem:

```
>>>> 1.0 // 0.1  
9.0 
```

The correct result should be 10.0 instead of 9.0 of course.

For now, we simply try to avoid this problem, but later implementation examples can use floating point numbers, too.

The // Operator on Floating-Points (2)

More Examples:

```
>>> 6.5 // 3.1
```

```
2.0
```

```
>>> -6.5 / 3.1
```

```
-3.0
```

```
>>> 6.5 // -3.1
```

```
-3.0
```

```
>>> -6.5 // -3.1
```

```
2.0
```

The // Operator on Decimal Fixed Points

In Addition to integer and floating-point operands, the operator // also works on decimal fixed point. To make it even more confusing, its behaviour on decimal fixed point slightly differs from its behaviour on floating-point...

Example:

```
>>> -7 // 4
```

```
-2 ←
```

```
>>> -7.0 // 4.0
```

```
-2.0 ←
```

```
>>> Decimal(-7) // Decimal(4)
```

```
Decimal('-1') ←
```

In contrast to floating-points, here we do not use the floor of the division result. Instead, we use the integer part of the quotient and then truncate towards zero.

This behaviour (see also % on decimal fixed point) better matches what we expect from textbook modular arithmetic:

$$a == (a // b) * b + a \% b$$

The Remainder (Modulo) Operator %

Remainder uses the operator: %

We already know modular arithmetic mod n , where $n \in \mathbb{N}$ and $n \geq 2$. For all $a \in \mathbb{Z}$, using Python's % operator, we simply write

$a \% n$

and Python computes $a \bmod n$.

Example:

```
>>>> 17 % 5
```

```
2
```

Python allows n to be a negative integer. In that case, the sign of the result is equal to the sign of the divisor (i.e. the second operand).

Moreover, $a \% 1 == 0$ and $a \% 0$ results in an error (division by zero).

Example:

```
>>>> 17 % -5
```

```
-2
```



The % Operator on Floating-Points (1)

The % operator also allows both operands to be floating-points. Then, for $x, y \in \mathbb{R}$: $x \% y$ is defined as

[the definition of this operator is left as homework]

Note that we also have to deal with possible rounding errors here.

Example:

```
>>>> 3.7 % 1.2
```

```
0.100000000000000031
```

Here, the correct result would have been 0.1 instead of 0.100000000000000031.

The % Operator on Floating-Points (2)

More Examples:

```
>>>> 3.7 % 1.2
0.100000000000000031
>>> 3.7 % -1.2
-1.0999999999999996
>>> -3.7 % 1.2
1.0999999999999996
>>> -3.7 % -1.2
-0.100000000000000031
>>>
```

Again, the minus-sign of the second operand defines the sign of the result.

The % Operator on Decimal Fixed Points

Again, the % operator also works on decimal fixed point. And again, its behaviour differs from floating-point operands:

Example:

```
>>> -7 % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
>>> Decimal(3.7) % Decimal(1.2)
Decimal('0.10000000000000003108624468950438313186168670654296875')
>>> Decimal(-3.7) % Decimal(1.2)
Decimal('-0.10000000000000003108624468950438313186168670654296875')
>>> Decimal(3.7) % Decimal(-1.2)
Decimal('0.10000000000000003108624468950438313186168670654296875')
>>> Decimal(-3.7) % Decimal(-1.2)
Decimal('-0.10000000000000003108624468950438313186168670654296875')
```

As we can see, this time the sign of the result matches the sign of the first operand instead of the second (as in the case of floating-points).

fmod(x,y) from math module

As we have seen, Python provides us with more than one option if we need to compute remainders of integers or floating-point numbers.

Already confused? OK, there is another one:

Python's math module contains the function **fmod**

fmod(x,y) is designed to conform to the following:

Let $r = \text{fmod}(x,y)$, then

$$x = k*y - r$$

with $r \in \mathbb{N}$, $0 \leq r < |y|$ and $k \in \mathbb{Z}$ such that the result r has the same sign as x .

fmod versus %

For more information about fmod, we include the following citation from Python's documentation:

*“Returns fmod(x, y), as defined by the platform C library. Note that the Python expression $x \% y$ may not return the same result. The intent of the C standard is that $\text{fmod}(x, y)$ be exactly (mathematically; to infinite precision) equal to $x - n*y$ for some integer n such that the result has the same sign as x and magnitude less than $\text{abs}(y)$. Python's $x \% y$ returns a result with the sign of y instead, and may not be exactly computable for float arguments. For example, $\text{fmod}(-1\text{e}-100, 1\text{e}100)$ is $-1\text{e}-100$, but the result of Python's $-1\text{e}-100 \% 1\text{e}100$ is $1\text{e}100-1\text{e}-100$, which cannot be represented exactly as a float, and rounds to the surprising $1\text{e}100$. For this reason, function `fmod()` is generally preferred when working with floats, while Python's $x \% y$ is preferred when working with integers.”*

divmod: Quotient and Remainder

divmod(a,b)

We can combine the two functions division and remainder in one function called `divmod`. `divmod` takes two (either integer or floating-point) numbers as arguments and returns two numbers.

If both operands are of type integer, then we get the expected behaviour:

`divmod(a,b) = (a // b, a % b)`

However, if one or both operands are floating-point, then the official Python documentation states the following:

*“With mixed operand types, the rules for binary arithmetic operators apply. For floating point numbers the result is $(q, a \% b)$, where q is usually $\text{math.floor}(a / b)$ but may be 1 less than that. In any case $q * b + a \% b$ is very close to a , if $a \% b$ is non-zero it has the same sign as b , and $0 \leq \text{abs}(a \% b) < \text{abs}(b)$.”*

This isn't exactly what we need... As we see, again we run into all sorts of rounding error-related problems. Therefore, we will refrain from using floating-point numbers here.

Efficient Computation Modulo 2^k

For some special cases with respect to the modulus n , there are faster alternatives than simply using $x \% n$. Here, we will only look at the case $n = 2^k$.

In Python, for $x \bmod 2^k$ we can write: $x \% 2^{**k}$.

It turns out that this is equivalent to $x \& (2^k - 1)$ where $\&$ stands for binary AND.

Example:

For $k = 1$ we get: $x \bmod 2 == x \& 1$

For $k = 2$ we get: $x \bmod 4 == x \& 3$

and so on...

The binary AND simply removes leading zeros and keeps the relevant rest.

Remark: some clever compilers detect special cases like this and replace the default operation by something more efficient.

Exponentiation in Python (1)

Suppose we have two operands x and y and we want to compute x^y . Python provides several options to do this.

- i. `x**y`
- ii. `pow(x,y[,z])`
- iii. `math.pow(x,y)`

ad (i) and (ii): If x and y are both integers and $y \geq 0$ then the result is also an integer. In that case, an optional third operand z is allowed and `pow(x,y,z)` returns x to the power of y , modulo z . This is more efficient than computing `pow(x,y) % z`.

If y is negative, then the result is floating-point.

Remark: please note that, for example, the term `-1**2` results in `-1` since Python evaluates the expression from right to left!

ad (iii): Python's `math` module provides `math.pow(x,y)` which always converts and delivers floating-points, even if x and y are both integers. Most functions in the `math` module deliver floating-point results.

Exponentiation in Python (2)

The Python 3.7 documentation states “*The two-argument form `pow(x,y)` is equivalent to using the power operator `x**y`.*” This implies that it doesn’t matter whether we use one or the other. Let’s do a reality check...

Python provides a `timeit` module that allows us to track the execution time of the above calculations. Let’s calculate 2 to the power of 1000, for example. For better accuracy, we repeat the calculation 1,000,000 times.

Example:

```
>>> import timeit
>>> timeit.timeit('2**1000', number=1000000)
1.2078252040082589
>>> timeit.timeit('pow(2,1000)', number=1000000)
1.2576940160943195
```

>>> In this example we can see that there is a slight difference between both options.

*Remark: during our lesson, this fact was presented with a bigger difference, due to a typo (I used `2*1000` instead of `2**1000`).*

Exponentiation in Python (3)

To compare the efficiency of two seemingly equivalent operations, we take a look at their respective byte code. Python provides us with the `dis` module which allows us to disassemble operations as follows.

Example:

```
>>> import dis
```

```
>>> dis.dis('2**1000')
```

```
1           0 LOAD_CONST           0 (2)
           2 LOAD_CONST           1 (1000)
           4 BINARY_POWER
           6 RETURN_VALUE
```

```
>>> dis.dis('pow(2,1000)')
```

```
1           0 LOAD_NAME            0 (pow)
           2 LOAD_CONST           0 (2)
           4 LOAD_CONST           1 (1000)
           6 CALL_FUNCTION          2
           8 RETURN_VALUE
```

Square Roots in Python (1)

Python offers several options for integer exponentiation. We need this quite often in public key cryptography. For some public key algorithms, we also need to compute square roots of large integers. How do we do this?

Why not just use $a**b$ with $b = \frac{1}{2}$? This returns a floating-point, not an integer.

We could simply convert the result back to an integer, but we still have to deal with floating-point precision, size limits and rounding issues. Let's look at some examples.

Example:

```
>>> a = 2**2048
```

```
>>> a**(1/2)
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#124>", line 1, in <module>
```

```
    a**(1/2)
```

```
OverflowError: int too large to convert to float
```

```
>>>
```

As we can see, large integers of typical “RSA-size” are too large for our floating-point.

Square Roots in Python (2)

Let's see what happens if we test for potential rounding issues.

Example:

```
>>> a = 10**100 - 1
>>> b = a*a
>>> b**(1/2)
1e+100
```

Even though we subtracted 1 from a , the return value looks like this never happened. Also remember that we get a floating-point instead of an integer result.

Thus, we still need a better solution for computing large integer square roots...

Square Roots in Python (3)

Now we look at Python's math library which contains a dedicated square root function called `math.sqrt`

Example:

```
>>> a = 10**100 - 1
>>> b = a*a
>>> import math
>>> math.sqrt(b)
1e+100
```

Obviously, this doesn't help either, since the result is floating-point again. For the same reason, we cannot compute with very large numbers:

```
>>> a = 2**2048
>>> math.sqrt(a)
Traceback (most recent call last):
  File "<pyshell#136>", line 1, in <module>
    math.sqrt(a)
OverflowError: int too large to convert to float
>>>
```

Square Roots in Python (4)

Finally, we try the decimal library in Python.

Example:

```
>>> from decimal import *
>>> a = 10**100 - 1
>>> b = a*a
>>> Decimal(b).sqrt()
Decimal('1.00000E+100')
```

Again, the result is wrong, since the default precision (28 places) is not sufficient. But this time, the decimal library allows us to increase the precision as follows:

[illegible]

Now we get the correct result, which is $a = 10^{**100} - 1$.

Converting Floating-Point to Integer

Python's `int()` function converts floating-point to integer. However, `int(x)` may either round or truncate $x \in \mathbb{R}$ and then return the corresponding integer. A more reliable alternative comes from Python's `math` module. There, we have three different options:

`math.trunc(x)` truncates the digits following the decimal

`math.floor(x)` returns the largest integer $\leq x$

`math.ceil(x)` returns the smallest integer $\geq x$

Freiwillige Aufgaben

- **Task 3.1**

Using other students' input from last week's homework results, - adjust, enlarge and improve your table from homework P.1.1 with respect to the three most important cryptographic libraries for Python.

- **Task 3.2**

Look into Python's documentation and research the different options to compute integer division and remainders in Python. Create a table and compare/describe the existing operators/functions at least with respect to the following: efficiency, type of return result, sign of return result, precision/exactness, etc.

- **Task 3.3**

The % operator also allows both operands to be floating-points. Then, for $x, y \in \mathbb{R}$, give an exact mathematical definition of the return result of $x \% y$

- **Task 3.4**

Try different options provided by Python to compute exponentiation and remainders modulo very large integers (very large means up to approx. 2048 bit long). More exactly, for some very large $a, b, n \in \mathbb{N}$ compute $a \bmod n$, $a^b \bmod n$. How far do we get if we try to compute a^b (without modulo) for large a and b ?

Use the timeit function to compare the efficiency of those operations.

- **Task 3.5 (Optional / not mandatory)**

Write a Python program that takes an integer a and an integer g , $0 < a < 10^6$, $g \geq 2$. The program is supposed to convert the integer a into its corresponding g -adic representation.

Input from terminal: a and g . Output to screen: g -adic representation.

Mathematische Grundlagen

Heutige Inhalte

- Motivation

Wir werden uns im Verlauf der Vorlesung Krypto II eine Weile mit Public Key Verfahren beschäftigen. Hierbei treffen wir immer wieder auf Verfahren, in denen Primfaktoren eine Rolle spielen. Deshalb wiederholen wir zuerst die wichtigsten Sachverhalte rund um Primzahlen, Primfaktoren und die zugehörigen algebraischen Strukturen.

- Inhalte

- zunächst wiederholen wir einige bereits bekannte Notationen
- einfache Sachverhalte zur Teilbarkeit ganzer Zahlen
- eindeutige ganzzahlige Division mit Rest
- eindeutige g -adische Darstellung ganzer Zahlen
- größte gemeinsame Teiler und Primzahlzerlegungen
- später folgen (teils als Wiederholung): Euklid, Fermat, Chinesischer Restsatz

Beweise, Beweise, Beweise ?

Vorbemerkung:

Wir werden einzelne Sachverhalte nicht nur verwenden, sondern auch beweisen. Nach Beendigung der einführenden Mathematikveranstaltungen spielen formale Beweise kaum noch eine Rolle im Studium. Das Beweisen ist jedoch eine wichtige Fertigkeit, die – wie alles andere auch – einer gewissen Übung bedarf. Wir werden Beweise zwecks leichter Nachvollziehbarkeit in jeweils *sehr* ausführlicher Form vortragen, mit deutlich mehr Zwischenschritten, als dies in Fachbüchern üblicherweise der Fall ist.

Manchmal sind Beweise zudem konstruktiv, in dem Sinne, dass aus ihnen ein Berechnungsverfahren für den jeweiligen Sachverhalt abgeleitet werden kann. Insgesamt werden wir in Kryptographie II aber nur wenige Beweise ausführen.

Notation

\mathbb{N} : die natürlichen Zahlen $\{0,1,2,\dots\}$

\mathbb{Z} : die ganzen Zahlen $\{\dots,-2,-1,0,1,2,\dots\}$

\mathbb{Q} : die rationalen Zahlen

\mathbb{R} : die reellen Zahlen

Den Quotienten zweier reeller Zahlen α, β mit $\beta \neq 0$ schreiben wir α/β bzw. $\frac{\alpha}{\beta}$.

Für $n \in \mathbb{N}$ schreiben wir statt $\frac{1}{\alpha^n}$ auch α^{-n}

Für $\alpha \in \mathbb{R}$ definieren wir $\lfloor \alpha \rfloor = \max \{a \in \mathbb{Z} : a \leq \alpha\}$. Bezeichnung: ***floor*** von α .
Ebenso $\lceil \alpha \rceil = \min \{a \in \mathbb{Z} : \alpha \leq a\}$. Bezeichnung: **ceiling** von α .

Beispiele: $\lfloor 7,46 \rfloor = 7$, $\lceil 7,46 \rceil = 8$, $\lfloor -7,46 \rfloor = -8$, $\lceil -7,46 \rceil = -7$

Teilbarkeit

Definition: seien $a, n \in \mathbb{Z}$. Man sagt a teilt n wenn es ein $b \in \mathbb{Z}$ gibt mit $n = ab$.

Für a teilt n schreiben wir auch $a \mid n$

Bemerkung: jede ganze Zahl a teilt 0, denn für $b=0$ gilt $a \cdot 0 = 0$.

Satz: seien $a, b, c, d, e, f, g \in \mathbb{Z}$. Dann gilt:

A. aus $a \mid b$ und $b \mid c$ folgt $a \mid c$

Beweis: $\exists f, g \in \mathbb{Z}$ mit $af=b$ und $bg=c$, folglich gilt $c = bg = (af)g = a(fg)$ mit $fg \in \mathbb{Z}$

B. aus $a \mid b$ folgt $ac \mid bc$

Beweis: $\exists f \in \mathbb{Z}$ mit $af = b$, folglich gilt $bc = (af)c = (ac)f$ mit $f \in \mathbb{Z}$

C. aus $c \mid a$ und $c \mid b$ folgt $c \mid da+eb$ für alle $e, b \in \mathbb{Z}$

Beweis: $\exists f, g \in \mathbb{Z}$ mit $cf = a$ und $cg = b$, folglich gilt $da+eb = d(cf) + e(cg) = c(df+eg)$ mit $(df+eg) \in \mathbb{Z}$

D. aus $a \mid b$ und $b \neq 0$ folgt $|a| \leq |b|$ für alle $a, b \in \mathbb{Z}$

Beweis: $\exists f \in \mathbb{Z}$ mit $af = b$, folglich gilt $|b| = |af| = |a| |f| \geq |a|$, da $|f| \geq 1$

E. aus $a \mid b$ und $b \mid a$ folgt $|a| = |b|$ für alle $a, b \in \mathbb{Z}$

Beweis: falls $b=0$, so folgt $a=0$ und die Behauptung stimmt. Nehmen wir also an $b \neq 0$, dann folgt aus obigem (D): $|a| \leq |b|$ und $|b| \leq |a|$, also $|a| = |b|$

Ganzzahlige Division mit Rest

Satz: seien $a, b \in \mathbb{Z}$ mit $b > 0$, dann existieren eindeutige $q, r \in \mathbb{Z}$ so, dass $a = qb + r$, wobei $0 \leq r < b$. Hierbei gilt $q = \lfloor \frac{a}{b} \rfloor$ und $r = a - qb$.

Beweis: Erinnerung: $q = \lfloor \frac{a}{b} \rfloor$ bedeutet per Definition, dass $q \leq \frac{a}{b} < q+1$.

ad (ii): Angenommen es gebe eine Lösung so dass $a = qb + r$ mit $0 \leq r < b$. Dann:

$$0 \leq r < b \Rightarrow 0 \leq \frac{r}{b} < 1 \Rightarrow 0 \leq \frac{a - qb}{b} = \frac{a}{b} - q < 1 \Rightarrow q \leq \frac{a}{b} < q+1 \Rightarrow q = \lfloor \frac{a}{b} \rfloor$$

\uparrow teilen durch b \uparrow $r = a - qb$ \uparrow addiere beidseitig q \uparrow Definition von $\lfloor a/b \rfloor$

Aus der Eindeutigkeit von q folgt sodann die Eindeutigkeit von r .

Bezeichnung: wir nennen q den *ganzzahligen Quotienten* und r den *Rest* der Division von a durch b . Wir schreiben $r = a \bmod b$

g-adische Darstellung ganzer Zahlen

Wir kennen bereits die Darstellung natürlicher Zahlen in Binärform, also als String aus Nullen und Einsen. Dies ist eine Sonderform (für den Fall $g = 2$) der sogenannten g-adischen Darstellung, für eine natürliche Zahl $g > 1$. Bevor wir einen Satz hierzu beweisen, schauen wir uns einige Beispiele an:

Beispiele für $g = 2$:

$$15 = 2^3 + 2^2 + 2^1 + 2^0$$

$$12 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$$

Beispiel für $g = 3$:

$$15 = 1 \cdot 3^2 + 2 \cdot 3^1 + 0 \cdot 3^0$$

Dies schreiben wir auch als $15 = \sum_{i=0}^2 a_i 3^i$ mit $(a_0, a_1, a_2) = (0, 2, 1)$.

Eine solche Folge (a_0, a_1, \dots, a_i) nennen wir **g-adische Entwicklung**. Für $g=2$ sprechen wir von **Binärentwicklung**.

Anmerkung (Notation): Sei $g > 1$ eine natürliche Zahl, dann bezeichnen wir für beliebiges $\alpha \in \mathbb{R}$ mit $\alpha > 0$ den **Logarithmus** von α zur Basis g : $x = \log_g \alpha \Leftrightarrow g^x = \alpha$

Eindeutigkeit der g-adischen Darstellung (1)

Satz: seien $a, g \in \mathbb{N}$ mit $a > 0$ und $g \geq 2$. Dann lässt sich a eindeutig darstellen in der Form $a = a_n g^n + a_{n-1} g^{n-1} + \dots + a_1 g + a_0$, wobei $a_n \neq 0$ und $a_i \in \mathbb{N}$, $0 \leq a_i < g$.

Dies ist die g-adische Darstellung von a , auch abgekürzt als $(a_n, a_{n-1}, \dots, a_0)$.

Beweis: Vorgehensweise: (i) Zum einen müssen wir zeigen, dass es eine Lösung gibt. (ii) Zum anderen müssen wir zeigen, wenn es eine Lösung gibt, so ist sie eindeutig.

ad (i): Induktion nach a . Für $a = 0$ und $a = 1$ wählen wir einfach ein passendes a_0 . Sei $a \geq 2$ und die Behauptung bereits bewiesen für alle natürliche Zahlen kleiner a .

Zu zeigen: dann gilt die Behauptung auch für a .

Wir führen eine ganzzahlige Division mit Rest aus wie folgt:

$$a = qg + r, \quad 0 \leq r < g, \quad q \geq 0.$$

Setze $a_0 := r$. Falls $q = 0$ ist, so erhalten wir sofort die passende Darstellung $a = a_0$.

Betrachten wir also den Fall $q > 0$: dann ist $q < qg \leq a$, also $q < a$ und nach Induktionsannahme:

$$q = b_m g^m + b_{m-1} g^{m-1} + \dots + b_1 g + b_0, \quad \text{wobei } b_m \neq 0 \text{ und } b_j \in \mathbb{N}, \quad 0 \leq b_j < g.$$

Eindeutigkeit der g-adischen Darstellung (2)

Wir schreiben nun $a = qg + r = qg + a_0 = (b_m g^m + b_{m-1} g^{m-1} + \dots + b_1 g + b_0) g + a_0 =$
 $= b_m g^{m+1} + b_{m-1} g^m + \dots + b_1 g^2 + b_0 g + a_0.$

Diese Darstellung hat die gewünschten Form, mit $a_{m+1} = b_m, a_m = b_{m-1}, \dots, a_1 = b_0.$

ad (ii): angenommen, die Darstellung sei nicht eindeutig. Dann finden wir

$$a_n g^n + a_{n-1} g^{n-1} + \dots + a_1 g + a_0 = a = a'_k g^k + a'_{k-1} g^{k-1} + \dots + a'_1 g + a'_0$$

wobei $a_n \neq 0$ und $a'_k \neq 0$. O.B.d.A. sei hierbei $k \geq n$.

Als erstes zeigen wir, dass $k = n$ sein muss. Nehmen wir an es sei $k \geq n+1$:

Es ist $a \geq g^n$ und ebenso $a \geq g^k$, ferner sind alle $a_i, a'_i \leq g-1$. Hieraus folgt

$$\begin{aligned} g^n \leq a &= a_n g^n + a_{n-1} g^{n-1} + \dots + a_1 g + a_0 \leq (g-1)g^n + (g-1)g^{n-1} + \dots + (g-1)g + (g-1) = \\ &= (g^{n+1} - g^n) + (g^n - g^{n-1}) + \dots + (g^2 - g) + g - 1 = \\ &= g^{n+1} - 1 < g^{n+1} \end{aligned}$$

Aus $k \geq n+1$ folgt $g^{n+1} \leq g^k \leq a < g^{n+1}$, ein offensichtlicher Widerspruch, weshalb k nicht größer als n sein kann. Folglich $k = n$.

Eindeutigkeit der g-adischen Darstellung (3)

Nun bleibt zu zeigen, dass die $a_i = a'_i$ sind für alle i . Angenommen, dies sei nicht der Fall. Dann sei j der größte Index mit $a_j \neq a'_j$ und $0 \leq j \leq n$.

Wir stellen zunächst fest, dass $j > 0$, da ansonsten $a_0 = a'_0$ und zugleich $a_0 \neq a'_0$.

Wir haben $a_j g^j + a_{j-1} g^{j-1} + \dots + a_1 g + a_0 = a'_j g^j + a'_{j-1} g^{j-1} + \dots + a'_1 g + a'_0 \Rightarrow$

$$\Rightarrow (a_j - a'_j)g^j + (a_{j-1} - a'_{j-1})g^{j-1} + \dots + (a_0 - a'_0) = 0$$

$$\Rightarrow (a_j - a'_j)g^j = -[(a_{j-1} - a'_{j-1})g^{j-1} + \dots + (a_0 - a'_0)] = (a'_{j-1} - a_{j-1})g^{j-1} + \dots + (a'_0 - a_0)$$

Wir nutzen nun die Tatsache, dass $|x+y| \leq |x| + |y|$ und $|xy| = |x| |y|$.

$$|(a_j - a'_j)| g^j \leq |(a'_{j-1} - a_{j-1})| g^{j-1} + \dots + |(a'_0 - a_0)|.$$

Da $|a_i - a'_i| \leq g-1$ für alle i , folgt hieraus:

$$|(a_j - a'_j)| g^j \leq (g-1)g^{j-1} + \dots + (g-1)g + (g-1) = g^j - g^{j-1} + g^{j-1} - g^{j-2} + \dots - 1 = g^j - 1.$$

Wegen $|(a_j - a'_j)| \geq 1$ ergibt dies den Widerspruch $g^j \leq |(a_j - a'_j)| g^j \leq g^j - 1$.

Folglich kann nicht $a_j \neq a'_j$ sein und die Behauptung ist bewiesen.

Binär- und Hexadezimalentwicklung

Für die Hexadezimaldarstellung natürlicher Zahlen verwenden wir die 15 Elemente der Menge {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, G}.

dezi mal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
hexa dez.	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
binär	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Haben wir die Binärentwicklung einer Zahl, so können wir diese einfach in Vierergruppen unterteilen und erhalten hieraus die Hexadezimaldarstellung, beispielsweise wie folgt:

11001011000100110101 = CB135
C B 1 3 5

In analoger Weise erhalten wir aus der Hexadezimaldarstellung die Binärdarstellung einer Zahl.

Binäre Länge einer Zahl

Bei der Betrachtung des Rechenaufwands zur Durchführung eines kryptographischen Algorithmus berücksichtigen wir zumeist die Größe des Inputs. Im Fall einer natürlichen Zahl bemisst sich dies in ihrer Länge. Da Computer Berechnungen binär kodiert ausführen, betrachten wir nicht die Anzahl der Dezimalstellen, sondern die Länge der Binärdarstellung.

Die Länge einer natürlichen Zahl a bezeichnen wir mit ***size(a)***. Dies ist die sogenannte ***binäre Länge*** von a .

Betrachten wir nicht nur natürliche sondern ganze Zahlen, so benötigen wir ein weiteres Bit für das Vorzeichen, so dass $\text{size}(a) = \text{size}(|a|) + 1$.

Bei Komplexitätsbetrachtungen spielt dieses eine Bit allerdings keine wesentliche Rolle. Hier interessieren wir uns nur dafür, ob mit der Inputlänge der Aufwand z.B. linear, logarithmisch, quadratisch, exponentiell etc. ansteigt. Mehr zu solchen Aufwandsbetrachtungen in einer anderen Vorlesung.

Größter Gemeinsamer Teiler

Definition: seien $a, b, c \in \mathbb{Z}$. Teilt c sowohl a als auch b , so heißt c ***gemeinsamer Teiler*** von a und b .

Satz: seien $a, b \in \mathbb{Z}$, $a \neq 0$. Dann hat die Menge der gemeinsamen Teiler von a und b ein (bezüglich \leq) größtes Element. Dieses heißt ***größter gemeinsamer Teiler*** von a und b . Es wird auch abgekürzt mit $\text{ggt}(a, b)$ oder $\text{gcd}(a, b)$.

Beweis: die Teiler von a sind größenmäßig beschränkt durch $|a|$.

Die gemeinsamen Teiler von a und b stellen eine Teilmenge der Teiler von a dar und sind folglich ebenfalls durch $|a|$ beschränkt. Daher muss die Menge der gemeinsamen Teiler ein größtes Element haben.

Anmerkung 1: per Definition setzt man $\text{ggt}(0, 0) = 0$.

Anmerkung 2: der größte gemeinsame Teiler ist nie negativ, denn falls $a \in \mathbb{Z}$ gemeinsamer Teiler, so ist auch $-a$ gemeinsamer Teiler, und eine der beiden Zahlen ist positiv und somit größer als die andere.

Euklidischer Algorithmus (1)

- Teilbarkeit und größter gemeinsamer Teiler spielen in der Kryptographie eine wesentliche Rolle. Die Berechnung des ggt ist für uns daher von großer Bedeutung.
- Standardverfahren zur Berechnung des ggt ist der Euklidische Algorithmus. Diesen lernt man oftmals schon in der Schule kennen, dann in der Mathematik-Vorlesung, spätestens in Krypto 1 erneut, und auch wir nehmen uns hier in Krypto 2 nochmals Zeit dafür, da der Euklidische Algorithmus in verallgemeinerter Form von uns immer wieder benötigt wird.
- Eigentlich handelt es sich um einen mathematischen Satz. Dessen Beweis erfolgt sodann konstruktiv in Form eines Algorithmus zur Berechnung des ggt.
- Wir werden anschließend auch den sogenannten Erweiterten Euklidischen Algorithmus nochmals betrachten.
- Hierauf aufbauend können wir dann weitere für uns wichtige Algorithmen einführen und besser verstehen.

Euklid

Euklid lebte der Überlieferung nach um 300 v. Chr. im ägyptischen Alexandria. Er befasste sich mit Arithmetik, Geometrie und Musiktheorie. Berühmtestes Werk: "Elemente", über Geometrie. Dieses wurde bis ins 20. Jahrhundert hinein als Lehrbuch verwendet.

Nach Euklid sind folgende mathematische Strukturen benannt (siehe Wikipedia):

- Euklidischer Abstand, die Länge der direkten Verbindung zweier Punkte in der Ebene oder im Raum
- **Euklidischer Algorithmus**, ein Verfahren zur Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen
- Euklidische Geometrie, die anschauliche Geometrie der Ebene oder des Raums
- Euklidischer Körper, ein geordneter Körper, in dem jedes nichtnegative Element eine Quadratwurzel besitzt
- Euklidische Norm, die Länge eines Vektors in der Ebene oder im Raum
- Euklidischer Raum, der Anschauungsraum, ein reeller affiner Raum mit dem Standardskalarprodukt
- Euklidische Relation, eine Relation, für die gilt: stehen zwei Elemente jeweils zu einem dritten in Relation, dann stehen sie auch zueinander in Relation
- Euklidischer Ring, ein Ring, in dem eine Division mit Rest möglich ist
- Euklidische Werkzeuge, die erlaubten Handlungen bei der Konstruktion mit Zirkel und Lineal

Zudem sind nach Euklid folgende mathematische Sätze und Beweise benannt:

- Euklids Beweis der Irrationalität der Wurzel aus 2, der erste Widerspruchsbeweis in der Geschichte der Mathematik
- Höhensatz des Euklid: In einem rechtwinkligen Dreieck ist das Quadrat über der Höhe flächengleich dem Rechteck aus den Hypotenusenabschnitten
- Kathetensatz des Euklid: In einem rechtwinkligen Dreieck sind die Kathetenquadrate jeweils gleich dem Produkt aus der Hypotenuse und dem zugehörigen Hypotenusenabschnitt
- Lemma von Euklid: Teilt eine Primzahl ein Produkt zweier Zahlen, dann auch mindestens einen der beiden Faktoren
- Satz von Euklid: Es gibt unendlich viele Primzahlen