Diskrete Mathematik **FTB-INF**
Dr. Martin E. Müller **WS 24/25**

Skriptum - Zahlen und modulare Arithemetik (v 1.1)

Dieses Skriptum begleitet die Vorlesung Diskrete Mathematik (Mathematik für Informatiker) an der Fachhochschule Vorarlberg. Es ist jedoch nicht vollständig und eignet sich daher (ohne Ergänzungen) nicht zur Vorbereitung auf die Prüfung. Beispiele, Beweise und zum Teil auch textuelle Ergänzungen werden während der Vorlesung gemeinsam erarbeiten.

Die Vorlesung folgt in weiten Teilen dem Buch *Konkrete Mathematik (nicht nur) für Informatiker* von Edmund Weitz [1]. Dieses Buch verfolgt den Ansatz, dass Informatiker*innen in ihrem späteren Berufsleben mehr angewandte (konkrete) Mathematik benötigen anstelle von „reiner“ Mathematik, welche Mathematik der Mathematik wegen betreibt und in der Lehre oft abstrakt und ohne nähere Betrachtung von Anwendungen bleibt. Unser Anwendungswerkzeug der Wahl: Programmieren! Sie lernen in unserer Vorlesung die mathematischen Begriffe, Methoden und vor allem deren Anwendungen, indem Sie diese in Computerprogramme „übersetzen“. Wir werden dafür (wie auch Herr Weitz) Python und JupyterLab verwenden.

Das Skriptum enthält mit hoher Wahrscheinlichkeit auch Fehler (typographisch sowie inhaltlich). Bitte melden Sie jeden Fehler der ihnen auffällt. Ihre Kommilitonen (und natürlich auch ich) werden es ihnen danken!

Natürliche Zahlen sind die Zahlen, die man zum Zählen benutzt. Die 0 ist hier nicht immer dabei, bei uns schon

$$\mathbb{N} = \{0, 1, 2, 3 \ldots\}.$$

Wir definieren hier $\mathbb{N}$ als **Menge** durch Aufzählung ihrer Elemente. Wenn Sie das hier noch nicht ganz verstehen, ist das nicht schlimm. Wir werden uns mit Mengen später ausführlich beschäftigen.

Ganze Zahl ist positive, natürliche Zahl, 0, oder das „negative Gegenstück“ einer positiven natürlichen Zahl (Minuszeichen vor positiver natürlicher Zahl).

$$\mathbb{Z} = \{\ldots, -5, -4, -3, -2, -1, 0, 1, 2, 3, \ldots\}.$$

Eine Zahl ist ein abstraktes Konzept! == man kann sie nicht sehen oder anfassen (nicht so wie Bäume, Häuser, …)

Was wir sehen können ist das *Symbol* „7“. Das ist jedoch nicht dasselbe! (Gleich wenig wie ein Foto von einem Baum dasselbe wie der Baum ist)

Neben dem Symbol „7“ existieren noch viele andere Wege, die Zahl 7 zu repräsentieren!

z.B.:

All diese Symbole repräsentieren die Zahl 7.

Durchgesetzt hat sich die Darstellung im *indisch-arabischen Dezimalsystem*, da es Berechnungen jeglicher Art stark erleichtert.

z.B.: $4358 + 215$

$\Rightarrow$ Wir müssen „nur“ die einzelnen „Ziffern“, nicht die ganze Zahl, addieren!

Auch Multiplizieren ist „einfach“!

z.B.: $4358 \cdot 215$

⇒ im Dezimalsystem können wir mit dem „kleinen Einmaleins“ „beliebig“ große Zahlen multiplizieren, weil dieses System ein **Stellenwertsystem** ist.

Zum Beispiel kann die Zahl 5338 geschrieben werden als

was insbesondere bedeutet, daß die erste „3“ etwas anderes bedeutet wie die Zweite (eine steht für 300, eine für 30), weil sie an einer anderen **Stelle** stehen.

⇒ Wir können mit zehn Symbolen beliebig große Zahlen schreiben (So wie wir mit 26 Buchstaben beliebig lange Texte schreiben können).

Die Zahl 10 spielt in diesem System eine prominente Rolle! Jede Position entspricht einer **Potenz** von 10. Das indisch-arabische Dezimalsystem ist also ein Stellenwertsystem mit der **Basis** 10 und wir brauchen hier nur 10 Ziffern ($\{0, 1, 2, \ldots, 9\}$) um beliebig große Zahlen darstellen zu können.

Computer verwenden als Basis die 2 (Strom und kein Strom) und man nennt dieses System **Binärsystem**.

Zum Beispiel hat in binärer Notation die Ziffernfolge „101010“ folgende Bedeutung:

Daraus können wir unseren ersten Algorithmus ableiten, der eine Zahl in Binärdarstellung in das Zehnersystem umrechnet:

1) Durchlaufe die Ziffern von links nach rechts. Beginne mit dem Zwischenergebnis 0
2) Addiere in jedem Durchlauf die entsprechende Ziffer zum Zwischenergebnis und verdopple anschließend außer im letzten Schritt

PYTHON: `bin2Dec`

Unser Algorithmus klammert die 2 „geschickt“ aus und wertet dabei von innen nach außen aus:

z.B.:

Auch Dezimal in Binär ist „leicht“, da

$13 =$

folgt, daß letzte Ziffer der Binärdarstellung **Rest** von Division durch 2 ist. Also im Beispiel $\frac{13}{2} = 6$, Rest **1**.

Addition in binärer Darstellung:

Multiplikation in binärer Darstellung:

$\Rightarrow$ Sehr einfach $\Rightarrow$ das ist der zweite Grund, warum unsere Computer Zahlen binär darstellen: die arithmetischen Operationen sind so einfach, daß man sie durch vergleichsweise „einfache“ Schaltkreise realisieren kann!

Diese Schaltkreise sind in der ALU (Arithmetic Logic Unit) realisiert:

Diese hat beschränkte Anzahl Bits um die Operanden darzustellen (heute üblicherweise 64). Wir verwenden im Folgenden 8 Bit zur Veranschaulichung der Prinzipien.

z.B. ist in 8-Bit-ALU $28_{10} + 9_{10}$

aber $228_{10} + 90_{10} =$

Da wir das erste Bit „verlieren“, liefert unsere 8-Bit-ALU

plus einen Hinweis, daß hier etwas nicht „stimmt“ (arithmetic overflow).

Diesem Verhalten widmet die Mathematik ein eigenes Teilgebiet: die **modulare Arithmetik** (Carl Friedrich Gauß $\sim$ 1800, Teil der Zahlentheorie)

Mit Hilfe des „Verschiebens“ (weglassen bzw. hinzufügen) von Bits können wir am Computer sogar Multiplizieren und Dividieren! Schauen wir uns dazu zuerst die Multiplikation bzw. Division im Gewohnten Stellenwertsystem (Dezimalsystem) an:

z.B. Multiplikation bzw. Division mit 100

Oder Division von 53861 durch 1000:

Wir müssen bei Multiplikation oder Division mit einer 10er Potenz (10^n) von Zahlen in dezimaler Darstellung lediglich 0-en anfügen (Multiplikation) bzw. „streichen“ (Division).

Das funktioniert im Binärsystem für 2-er-Potenzen ($10_2 = 2_{10}$, $100_2 = 4_{10}$, $1000_2 = 8_{10}$, ...) gleich!

z.B. ist $110011_2 \cdot 1000_2$

und $1111000111_2/10000_2$

Dies ist nicht nur für uns sondern auch für den Computer einfach, da (wie schon erwähnt) nur Bits „verschoben“ werden müssen

⇒ **Bit Shift!** In PYTHON werden Bits mit dem $\gg$-Operator nach rechts (z.B. liefert $1010_2 \gg 2\ 10_2$) und mit dem $\ll$-Operator nach links verschoben (z.B. liefert $10_2 \ll 2\ 1000_2$)

Bsp: $1000 \gg 2 \ll 2$ und $1111 \gg 2 \ll 2$

Was passiert bei $23_{10} \cdot 27_{10}$?

Wenn wir „nur“ die „rechten“ 8-Bits betrachten entspricht dies

was einer Subtraktion von 256 ($= 2^8$) entspricht, bis das Resultat in acht Bits „reinpasst“.

Dies entspricht in Basis 10

was in 8-Bit-Arithmetik 109 ist (wir haben 256 2× abgezogen). Auf einer 64-Bit-Maschine ziehen wir 2^{64} von einem Resultat ab, bis es in 64 Bit „reinpasst“. Allerdings müssen wir uns hier nicht auf 2-er Potenzen beschränken!

Bei 8 Bit „akzeptieren“ wir nur Zahlen ≤ 256. Was passiert, wenn wir nur Zahlen ≤ 12 akzeptieren?

Dann wäre $9 + 5$

Dies ist das „Wesen“ der **modularen Arithmetik**: Sie rechnen wie auf einem Zifferblatt!

Unser 8-Bit-Computer hat dann 256 Stunden und nach der 255 kommt wieder die 0. Arbeiten wir mit einem „normalen“ Zifferblatt (12 Stunden) rechnen wir **modulo** 12.

z.B. ist $5 + 9 =$

und $23 \cdot 27 =$

In PYTHON ist der modulo Operator das `%`-Zeichen:

Rechnen wir z.B. modulo 12, dann behandeln wir die 19 wie die 7 da $19 \mod 12 = 7$ ist und wir schreiben

und sagen „7 und 19 sind kongruent modulo 12“.

> Allgemein sind zwei Zahlen a und b kongruent modulo n, wenn sie beide denselben Rest bei Division durch n haben.

Dies ist gleichbedeutend mit der Bedingung, dass der Abstand $|a - b|$ durch n teilbar sein muss.

z.B. sind 107 und 73 kongruent modulo 17?

Zwei Zahlen a und $b = a + n \cdot 17$ (ihr Abstand ist ein Vielfaches von 17) sind modulo kongruent 17.

z.B. $a = 25$, $b = a + (3 \cdot 17)$

$\Rightarrow$ Zahlen die modulo kongruent sind können auf dem Zahlenstrahl als Punkte gleichen Abstands visualisiert werden:

$\Rightarrow$ 2 und 7 sind kongruent modulo 5, weil ihr Abstand $1 \cdot 5$ ist. $7 \equiv 22$ weil ihr Abstand $15 = 3 \cdot 5$ usw. und $2 \equiv 2$ da Abstand $0 = 0 \cdot 5$!

PYTHON: `modulo_congruent`

Warum ist das für Berechnungen in endlicher Arithmetik (endliche Anzahl Bits der ALU) so wertvoll?

$\Rightarrow$ Wenn wir modulo einer festen Zahl n rechnen, spielt es keine Rolle, welchen „Repräsentanten“ wir wählen, wir bekommen immer (modulo n) dasselbe Ergebnis!

Rechnen z.B. modulo 6 bedeutet Rechnen in $\mathbb{Z}/6\mathbb{Z}$ wobei $\mathbb{Z}/6\mathbb{Z}$ die Menge aller möglichen Reste bei Division durch 6 ist (also $\{0, 1, 2, 3, 4, 5\}$).

Solch eine Menge (zusammen mit Addition und Multiplikation) nennt man **Restklassenring.**

$\Rightarrow$ Es gibt nur endlich viel „Dinge“ die man ausrechnen kann.

z.B. ist $4 \cdot 5$ in $\mathbb{Z}/6\mathbb{Z}$

$\Rightarrow$ Man kann für Restklassenringe Tabellen für $+$ und $\cdot$ erstellen und muß dann (für diesen Restklassenring) nie wieder Rechnen!

z.B. Additions- und Multiplikationstabelle für Restklassenring $\mathbb{Z}/4\mathbb{Z}$ ($\{0, 1, 2, 3\}$)

Wir rechnen hier alles modulo 4!

Zwei (erste) Anwendungen

- Teilbarkeitsprüfung durch 9
- Prüfung von Multiplikation

Teilbarkeitsprüfung:

Ist 9652 durch 9 ohne Rest teilbar? Dies können wir prüfen, indem wir prüfen, ob deren Quersumme (Summe der Ziffern) durch 9 teilbar ist. Der Grund dafür finden wir in der modularen Arithmetik. Rechnet man modulo 9, ist die Quersumme immer kongruent zu der Zahl selbst!

$\Rightarrow$ rechter Term kann modulo 9 „ignoriert“ werden. Dies liefert die iterierte Quersumme (Quersumme bis einstellig).

z.B. iterierte Quersumme von 9652

Prüfen von Multiplikation:

Ist $789234077 \cdot 457239331 = 360868851369882487$ richtig?

Ja, genau dann wenn (g.d.w.) die iterierte Quersumme der linken und rechten Seite identisch ist!

Aus 789234077 wird

aus 457239331 wird

und somit

Für die rechte Seite

$\Rightarrow 2 \neq 1 \Rightarrow$ stimmt nicht!

Bisher „nur“ Addition und Multiplikation, nun Erweiterung auf Subtraktion und Division. Diese sind „lediglich“ Umkehrung von $+$ und $\cdot$.

z.B. ist der Ausdruck $10 - 3$ Antwort auf die Frage, was zu 3 addiert werden muß, um in Summe 10 zu erhalten. Dies entspricht der Lösung der Gleichung $x + 3 = 10$. Man kann aber auch sagen, $10 - 3$ bedeutet dasselbe wie $10 + (-3)$. Aber was genau ist (-3)?

Die Zahl **Null** spielt bei der Addition eine besondere Rolle, da $a + \text{Null} = a$ und man nennt die Null das neutrale Element der Addition. Jede Zahl hat auch ein inverses Element bezüglich der Addition.

$\Rightarrow$ Summe einer Zahl und ihren inversen Elements ist Null. z.B. ist -8 das inverse Element der 8 bezüglich der Addition da $8 + (-8) = 0$

Die Existenz von inversen Elementen ist die Grundlage für die Lösung von Gleichungen! Will man die Gleichung $3 + x = 10$ lösen, muß man die 3 auf der linken Seite „loswerden“ um das x zu „isolieren“ ($==$ das x „freistellen“). Dies ist gleichbedeutend mit dem Ersetzen der 3 durch die Null (neutrales Element bezüglich der Addition), da $0 + x = x$. Dies können wir durch Subtraktion des inversen Elements der 3 ($== -3$) auf beiden Seiten der Gleichung erreichen, also z.B.

und so haben wir den Wert von x bestimmt, für den unsere Gleichung erfüllt ist.

Wie können wir inverse Elemente in Restklassenringen finden?

Was ist z.B. das inverse Element von 8 in $\mathbb{Z}/13\mathbb{Z}$? Auch hier müssen wir die Gleichung $8 + x = 0$ lösen (das x in unserem Restklassenring bestimmen, so daß die Gleichung „stimmt“). In unserem Restklassenring ist dies gleichbedeutend mit dem Lösen der Gleichung $8 + x = 13$, da $0 \equiv 13$ in $\mathbb{Z}/13\mathbb{Z}$ oder anders gesagt 13 in $\mathbb{Z}/13\mathbb{Z}$ eine Zahl ist, die durch die 0 repräsentiert wird.

also

und somit ist 5 das inverse Element bezüglich der Addition von 8 in $\mathbb{Z}/13\mathbb{Z}$.

Kommen wir nun zur Division und stellen uns zuerst die Frage, wann eine Zahl durch eine andere Teilbar ist. Eine Zahl b ist teilbar durch eine Zahl a ($==$ a teilt die Zahl b ganzzahlig (ohne Rest)) g.d.w. b ein Vielfaches von a ist. D.h. man kann eine Zahl k finden, so dass man b als $k \cdot a$ schreiben kann.

Z.B. teilt 5 die Zahl 20, da $20 = 4 \cdot 5$. Auch -5 teilt die 20 da $20 = (-4) \cdot (-5)$. Um einen Teiler zu finden, müssen wir also im Beispiel das k in der Gleichung $k \cdot 5 = 20$ und ganz allgemein das k in der Gleichung $k \cdot a = b$ bestimmen.

Frage: Teilt z.B. 7 die 0?

Ist 7 auch durch 0 teilbar?

$\Rightarrow$ 0 teilt überhaupt keine Zahl außer sich selbst. Jede Zahl ist jedoch Teiler der 0.

Ist b durch a teilbar schreibt man $a \mid b$ (a teilt b, a ist Teiler von b), ist a kein Teiler von b $a \nmid b$.

z.B.:

Ohne negative Zahlen ist der Rest einer modulo Operation der kleinstmögliche „Korrekturterm“. z.B. gilt $12 \mod 5 = 2$ und nicht 7. Durch Einführen von negativen Zahlen gilt diese Definition nicht mehr, da dann 12 auch als $(3 \cdot 5) - 3$ geschrieben werden kann (ohne negative Zahlen haben wir $(2 \cdot 5) + 2$) und $-3 < 2$ gilt.

Lösung: Wir (und fast alle anderen (Java, `C#`, aber Python nicht) lassen nur positive Divisionsreste zu

$\Rightarrow$ Der Rest einer Division ist eindeutig bestimmte nicht negative Zahl, die kleiner als der

Betrag des Divisors ist.

$\Rightarrow 12 \mod (-5) = \ldots$

$\Rightarrow$ Erweiterung unserer „modulo-Geraden“

$\Rightarrow$ Wir können immer noch zwischen „normaler“ und modularer Arithmetik „hin- und her-springen“!

Z.B. Inverse von 4 in $\mathbb{Z}/11\mathbb{Z}$ und $\mathbb{Z}$ bezüglich Addition

Dies gilt auch für die Subtraktion!

Z.B. ist $8 - 5$ in $\mathbb{Z}/11\mathbb{Z}$ ...

Und wie können wir nun in $\mathbb{Z}/n\mathbb{Z}$ dividieren? Dazu schauen wir uns zuerst an, wie wir den größten gemeinsamen Teiler (ggT) zweier Zahlen a und b bestimmen können: Dies wird auch heute noch mit dem wahrscheinlich ältesten Algorithmus der Welt ($\sim$ 2300 Jahre) gemacht: dem Algorithmus von Euklid!

Wir wissen bereits: b ist Teiler von a ($b \mid a$ == a ist durch b ganzzahlig teilbar) g.d.w. $a = k \cdot b$ geschrieben werden kann.

$\Rightarrow$

- ▷ Wenn a und b positiv, muß auch k positiv sein
- ▷ b kann nicht größer a sein

$\Rightarrow$ Jeder Teiler von a kann höchstens so groß wie a sein.

- $\triangleright$ 1 und -1 sind immer Teiler einer beliebigen Zahl $\in \mathbb{Z}$

Ein gemeinsamer Teiler zweier Zahlen a und b ist eine Zahl die beide teilt und wir wissen nun, daß

- $\triangleright$ immer mindestens einer existiert
- $\triangleright$ keiner größer als die kleinere der beiden Zahlen sein kann

$\Rightarrow$ es existiert auch immer ein ggT und man schreibt für ihn $\text{ggT}(a, b)$ (oder in englisch $\gcd(a, b)$)

Allererster (naiver) Algorithmus zur Bestimmung des ggT:

- $\triangleright$ starte mit kleineren Zahl und verringere sie um 1 bis sie ggT ist

$\Rightarrow$ sehr ineffizient, Umsetzung in Übung.

Nun verbessern wie diesen etwas „plumpen“ Algorithmus (== machen in effizienter). Dies ist eine sehr übliche Vorgehensweise in der Informatik. Oft sind sehr komplexe Algorithmen das Resultat Jahrelanger, iterativer Verbesserungen die mit einer sehr einfachen Idee gestartet haben (siehe z.B. Quadraturalgorithmus im zweiten Semester).

Dazu verwenden wir folgende Überlegung:

Ist c Teiler von a und b, dann teilt c auch $a + b$ sowie $a - b$.

Z.B: ist 3 Teiler von 42 und 66, da ...

Dies ist so, da $a \mod c = 0$ und auch $b \mod c = 0$, also

Allgemein gilt:

> Wenn c Teiler von a und b ist, dann teilt c auch jeden Term der Form $\alpha \cdot a + \beta \cdot b$, wobei α und β beliebig (auch nicht positive) ganze Zahlen sein können.

Beweis:

Damit können wir unseren Algorithmus schon wesentlich verbessern. Suchen wir z.B. den ggT von 400 und 225, können wir auch den ggT von 255 und 175 suchen, da $400 - 225 = 175$. Suchen wir den ggT von 255 und 175, ist dieser gleich dem ggT von 175 und 50 (da $225 - 175 = 50$) usw.

$\Rightarrow$ Wir ersetzen einfach immer wieder den größeren der beiden Werte durch deren Differenz:

Dieser Algorithmus endet, wenn beide Zahlen gleich sind und diese Zahl ist dann der ggT.

PYTHON: `ggT`

Ein weiterer Verbesserungsschritt besteht darin, nicht mehrfach zu subtrahieren (siehe Übung).

Wir haben bereits gezeigt, daß c jeden Term der Form $\alpha \cdot a + \beta \cdot b$ teilt, wenn c Teiler von a und b ist. Terme dieser Form nennt man **Linearkombination** von a und b. Da der ggT a und b teilen muß, muß er auch deren Linearkombinationen teilen!

$\Rightarrow$ Wir können den ggT von a und b als

schreiben und der Algorithmus von Euklid kann uns das α und β liefern, wenn wir nach Berechnung des ggT wieder „rückwärts“ gehen: z.B. „entstand“ der ggT 25 von 400 und 225 im letzten Schritt aus $50 - 25$, also

PYTHON: `extggT(a,b)`

Damit können wir nun (endlich) die Division in $\mathbb{Z}/n\mathbb{Z}$ umsetzen. Das neutrale Element der Division ist die 1 (da $a \cdot 1 = a$). Die Aufgabe der Division besteht darin, das multiplikativ Inverse (== Kehrwert) zu finden. Also das x in der Gleichung $a \cdot x = 1$. Suchen wir das in $\mathbb{Z}/n\mathbb{Z}$, rechnen wir modulo n. Also suchen wir $a \cdot x \mod n = 1$ und dies entspricht der Gleichung

für $y \in \mathbb{Z}$. Wir können $a \cdot x$ ja als jeden beliebigen Repräsentanten in $\mathbb{Z}/n\mathbb{Z}$ darstellen. (In $\mathbb{Z}$ bedeutet die Gleichung: Wir suchen ein Vielfaches von a das bei Division durch n 1 ergibt.)

Durch Umformung erhalten wir

wobei x und y gerade jene Werte aus dem erweiterten Eukliedschen Algorithmus sind!

z.B.: Kehrwert von 5 in $\mathbb{Z}/6\mathbb{Z}$

Das geht leider nicht immer. z.B. `extggT(4,6)`:

Der ggT von 4 und 6 ist also 2. 4 und 6 sind also nicht teilerfremd
$\Rightarrow$ Es gibt keinen Kehrwert von 4 in $\mathbb{Z}/6\mathbb{Z}$

Für welche Restklassenringe ($\mathbb{Z}/n\mathbb{Z}$) können wir immer (für alle Zahlen) Kehrwerte finden? Für die, in denen alle Zahlen „unterhalb“ von n teilerfremd sind $\Rightarrow$ falls n eine **Primzahl** ist!

Ist n prim schreibt man statt $\mathbb{Z}/n\mathbb{Z}$ „einfach“ $\mathbb{Z}_n$ und nennt eine solche Struktur einen **endlichen Körper**.

Allgemein (und informell) ist ein endlicher Körper eine Struktur, in der die vier Grundre-

chenarten so formuliert sind, wie wir es gewohnt sind. Bitte studieren Sie für eine formal korrekte Definition bei Interesse ein beliebiges Buch über algebraische Strukturen.

Warum brauchen wir als Informatiker endliche Körper? Da es für endliche Körper viele Anwendungen in der Informatik gibt! z.B. können wir mit endlichen Körpern Fehler in Nummern erkennen, wenn diese durch eine Prüfziffer „abgesichert“ sind.

Die ISBN-Nummer (International Standard Book Nummer), welche Bücher weltweit eindeutig identifiziert, verwendet z.B. den endlichen Körper $\mathbb{Z}_{11}$ um zu prüfen, ob eine gegebene Nummer eine gültige ISBN-Nummer ist, indem sie die letzte ihrer 10 Ziffern als Prüfnummer berechnet:

z.B.: Ist $1 - 4842 - 1177 - 4$ eine gültige ISBN-Nummer?

PYTHON: `checkISBN(digits)`

Welche „Eingabefehler“ können wir mit diesem Verfahren finden? Was passiert, wenn wir z.B. zwei Ziffern vertauschen? Lassen Sie uns anschauen, was passiert, wenn wir in unserem Beispiel von oben die zweite mit der dritten Ziffer vertauschen:

Da wir in $\mathbb{Z}_{11}$ rechnen und wir so immer ein Produkt zweier Zahlen $\neq 0$ haben, ist dieses Produkt immer $\neq 0$ (außer wir haben zwei Nullen vertauscht was aber egal wäre) und somit die Prüfsumme $\neq 0$.

$\Rightarrow$ Solche Fehler finden wir immer!

Dieses Prüfverfahren findet auch falsche Eingabe einer Ziffer (siehe Übung) sowie eine Fehlerhafte Prüfziffer (siehe Literatur).

Auch IBAN's (International Banking Account Number) verwenden ein Ähnliches Verfahren. Allerdings mit zwei Prüfziffern und in $\mathbb{Z}_{97}$. Die Idee hinter der Prüfung der Nummer auf Korrektheit ist allerdings ident mit der bei ISBN-Nummern, nur ein wenig aufwendiger zu rechnen. Weitere Anwendungen endlicher Körper werden wir später in diesem Abschnitt kennen lernen. Dazu benötigen wir aber zuerst noch ein wenig mehr Theorie, im speziellen eine ordentliche Definition von **Primzahlen**.

Primzahlen

Primzahlen waren lange ohne praktische Anwendung und daher ein theoretisches Konstrukt, welches im Teilgebiet der Zahlentheorie rein „mathematisch“ behandelt wurde. Heute wären viele technische Anwendungen wie zum Beispiel das sichere Übertragen von Nachrichten (Verschlüsselung von Nachrichten == Kryptographie) ohne Primzahlen und die geleistete Vorarbeit vieler Mathematiker nicht möglich.

Eine Primzahl ist eine Zahl größer 1, die außer der 1 und sich selbst keine weiteren positiven Teiler hat.

$\Rightarrow$ eine Primzahl ist immer positiv $\Rightarrow \in \mathbb{N}$.

PYTHON: `isPrime`, `getPrimes(n)`

Der erste Wurf zur Bestimmung der ersten n Primzahlen ist sehr ineffizient, da wir für jede Zahl die Methode `isPrime` aufrufen und `isPrime` im schlimmsten Fall für eine Zahl n wieder n Schleifendurchläufe benötigt ($\mathcal{O}(n^2)$ - siehe Komplexitätstheorie)

Das geht natürlich besser!

Wir müssen z.B. nicht alle Zahlen von $1 \dots n$ prüfen um festzustellen, ob n prim ist, da falls n nicht prim ist, n als $n = a \cdot b$ geschrieben werden kann, wobei $n > a, b > 1$. Wären a und b beide größer als $\sqrt{n}$, dann wäre $a \cdot b$ größer $\sqrt{n} \cdot \sqrt{n} = n$

$\Rightarrow$ Es kann nur einer der Teiler (a oder b) größer $\sqrt{n}$ sein
$\Rightarrow$ Es reicht, wenn `isPrime` bei der Suche nach potentiellen Teilern bis $\sqrt{n}$ geht $\Rightarrow$ Übung.

Dieser erste primitive Ansatz zur Prüfung ob eine Zahl prim ist benötigt für eine Zahl mit 100 Dezimalstellen (was in der Kryptographie immer noch wenig ist) $\sqrt{n} \sim 10^{50}$ Schleifendurchläufe! Wenn ein Computer in einer Sekunde 10^{15} Durchläufe schaffen würde, würde er $10^{50} - 10^{15} = 10^{35}$ Sekunden zur Prüfung benötigen. Ist das viel? Sehr viel sogar! Das Universum ist ~ 14 Milliarden Jahre alt $\leq 5 \cdot 10^{17}$ Sekunden.

$\Rightarrow$ wir müssen viel effizienter werden! $\Rightarrow$ Das Sieb des Eratosthenes

Jede Zahl größer 1 und nicht prim muß mindestens einen **Primteiler** (Teiler der Primzahl ist) haben.

Man kann sogar jede Zahl solange in Teiler „zerlegen“, bis nur noch Primzahlen übrig bleiben.

z.B. ist 4200 nicht prim und kann folgendermaßen zerlegt werden:

Mit diesem Verfahren kann man jede positive ganze Zahl als Produkt von Primzahlen darstellen und es gibt (abgesehen von der Reihenfolge der Multiplikatoren) genau eine Möglichkeit dies zu tun.

Dies ist der **fundamentalsatz der Arithmetik**!

Jede Zahl hat eine **kanonische Primfaktorzerlegung** die man erhält, wenn man die Primfaktoren sortiert und solche, die mehr als einmal vorkommen als Potenzen schreibt.

Für 4200 ist dies

⇒ Wir können alle ganzen Zahlen aus Primzahlen „aufbauen“!

PYTHON: `PrimeFactors`

Umgekehrt können wir eine Liste aller Primzahlen bis n erzeugen, indem wir mit einer Liste mit allen Zahlen starten und aussieben:

PYTHON: `pzSieb`

Auch so ist die Prüfung auf Primeigenschaft einer großen Zahl (ist eine große Zahl eine Primzahl) nicht effizient möglich und das ist auch gut so! Wenn es ein effizientes Verfahren zur Prüfung einer großen Zahl auf deren Primeigenschaft geben würde, würden heutige Kryptographieverfahren nicht funktionieren.

Die derzeit besten (effizientisten) Verfahren zur Primzahl-Prüfung beruhen auf probabilistischen Verfahren (siehe z.B. Weitz, Kapitel 9).

RSA-Kryptosystem (Rivest, Shamir, Adelman):

Die Verschlüsselung von Nachrichten wurde schon im Altertum praktiziert, bis in die 1970er-Jahre jedoch nur mit symmetrischer Verschlüsselung == Sender und Empfänger verwenden den gleichen Schlüssel für Ver- und Entschlüsselung.

⇒ Beide müssen den Schlüssel kennen, hat ihn ein „dritter“, kann er alle Nachrichten entschlüsseln.

- ▷ Bei weiten Entfernungen muss der Schlüssel übertragen werden ⇒ kann abgefangen werden
- ▷ man braucht für unterschiedliche Kommunikationspartner unterschiedliche Schlüssel.

Ein Beispiel für eine symmetrische Verschlüsselung ist die **Cäsar Verschlüsselung**:

Zuordnung von Zahl zwischen 0 und 25 zu Buchstaben. Schlüssel ist Zahl k zwischen 1 und 25. Verschlüsselung durch Ersetzen von Buchstabe m mit $m + k \mod 26$.

PYTHON: `caesar(msg, key)`

Zwei weitere Nachteile dieser Verschlüsselung:

- ▷ Es gibt nur 26 mögliche Schlüssel
  ⇒ man kann „probieren“
- ▷ Nur Buchstaben ersetzen ist leicht durch statistische Verfahren knackbar. z.B. ist e häufigster Buchstabe in deutschen Texten.

⇒ Wir brauchen was Besseres! == asymmetrische Verfahren

Das RSA-Kryptosystem ist ein asymmetrisches Verschlüsselungsverfahren bei dem Ver- und Entschlüsselung lediglich 3 Zahlen n, e (für encrypt) und d (für decrypt) benötigt

werden. Eine Nachricht N (ebenfalls eine Zahl) wird mit $N^e \mod n =: S$ verschlüsselt ($\Rightarrow N$ muß $< n$ sein). Der Empfänger entschlüsselt die verschlüsselte Nachricht S mit $M = S^d \mod n$.

d ist der private Schlüssel von Bob, e der öffentliche Schlüssel den er jedem geben kann. e und d werden folgendermaßen berechnet:

1) Berechne 2 Primzahlen p und q

2) $n = p \cdot q$

3) Berechne $\varphi(n) = (p-1)\cdot(q-1)$ und suche eine Teilerfremde Zahl zu $\varphi(n)$. Z.B. durch Primfaktorzerlegung von $\varphi(n)$ und Bilden des Produkts von „einigen“ Primzalhlen die alle nicht Teiler von $\varphi(n)$ sind. Das ist dann unser e.

4) d ist dann der Kehrwert von e modulo $\varphi(n)$. In PYTHON liefert das die Methode `mod_inverse(e, phi(n))`

z.B. Finden eines e´s für zwei „kleine“ Primzahlen

Wir können nun mit einer PYTHON-Funktion `crypt(M, k)`: und den drei Werten e, d, und n Nachrichten Ver- und Entschlüsseln:

PYTHON: `crypt(M, k)`:

Warum können wir mit `crypt` Ver- und Entschlüsseln? == Warum gilt

$$(N^e)^d = N$$

wenn man modulo n rechnet?

Rationale Zahlen

Computer unterscheiden intern zwischen ganzen Zahlen und Zahlen mit Dezimalpunkt (oder Exponent). Mischen wir ganze Zahlen mit Dezimalzahlen, z.B. $42 + 42.0$ wird daraus eine Dezimalzahl, also im Beispiel 84.0.

PYTHON: dec

- ▷ PYTHON rechnet „richtig“, wenn wir nur ganze Zahlen verwenden.
- ▷ Außerhalb der ganzen Zahlen (beim Rechnen mit Fließkommazahlen) sind Fehler kaum zu vermeiden!

Und wozu brauchen wir überhaupt mehr wie die ganzen Zahlen? Mit den natürlichen Zahlen können wir die Gleichung

$$5 + x = 2$$

nicht lösen. Dazu brauchen wir die ganzen Zahlen (negative Zahlen).

Allerdings können wir die Gleichung

$$5 \cdot x = 2$$

damit nicht lösen:

Dazu benötigen wir die **rationalen Zahlen**.

Ein Bruch hat die allgemeine Form a/b oder $\frac{a}{b}$, wobei $a \in \mathbb{Z}$ und $b \in \mathbb{Z}$ und $b \neq 0$.

$\Rightarrow \frac{a}{b}$ ist die Zahl, die die Gleichung $x \cdot b = a$ löst.

Somit hat jede Zahl außer der Null einen Kehrwert und man sagt für $\frac{a}{b}$ auch „a geteilt durch b“.

Insbesondere ist $\frac{1}{b}$ immer der Kehrwert von b.

Die Menge aller rationalen Zahlen wird mit $\mathbb{Q}$ (von Quotient) bezeichnet.

Mit dieser Definition ist auch $42 = \frac{42}{1}$ eine rationale Zahl und $\mathbb{Z}$ eine Teilmenge (mehr dazu später) von $\mathbb{Q}$ und $\mathbb{N}$ eine Teilmenge von $\mathbb{Z}$:

Brüche können unterschiedlich dargestellt werden. Z.B. ist $\frac{20}{30} = \frac{4}{6} = \frac{-4}{-6}$. Jeder Bruch hat jedoch eine eindeutige **kanonische Darstellung**. Das „Umwandeln“ eines Bruches in seine kanonische Darstellung nennt man **Kürzen**.

PYTHON: `lowestTerm`

Das Rechnen mit rationalen Zahlen kann wiederum auf das Rechnen mit ganzen Zahlen zurückgeführt werden:

Addition:

Multiplikation:

Subtraktion und Division werden auch hier wiederum als Kehrwerte berechnet. Dies dürfen Sie in der Übung implementieren (vergessen Sie dabei nicht auf das Kürzen!).

PYTHON: `add`

Brüche können auch im Dezimalsystem dargestellt werden. Z.B. ist $\frac{6}{10} = 0.6$.

Das entspricht einer Ausweitung unseres Dezimalsystems (Stellenwertsystems) auf negative Zehnerpotenzen!

z.B. ist 53.38

Manche Brüche können jedoch nur mit unendlich vielen Nachkommastellen korrekt dargestellt werden.

z.B. 1/3

Dies entspricht im Stellenwertsystem einer „unendlichen Summe“:

Dies ist eine **Reihe** (siehe zweites Semester Analysis)

Umwandeln eines Bruchs in eine Dezimalzahl: Falls Nenner auf Zehnerpotenz erweiterbar ist dies sehr einfach! z.B.:

Falls nicht, dividieren wir und hängen dem Zähler Nullen an

und suchen so die Periode.

Auch die Umwandlung einer Dezimalzahl in einen Bruch ist „einfach“:

1) Zähle Anzahl (#) Nachkommastellen und multipliziere Zähler und Nenner mit dieser Anzahl:

2) Kürze das Resultat

Wo liegen die rationalen Zahlen auf dem Zahlenstrahl?
$\Rightarrow$ „fast“ überall!

Ausgehend von zwei ganzen Zahlen $a, b \in \mathbb{Z}$ können wir immer eine Zahl $c \in \mathbb{Q}$ mit $c = \frac{a+b}{2}$ bilden, die in der Mitte zwischen a und b liegt. Auf die gleiche Art können wir eine Zahl $d = \frac{c+a}{2}$ bilden, welche wiederum zwischen c und a liegt usw.:

Zwischen den einzelnen rationalen Zahlen gibt es somit (fast) keinen Platz

== sie liegen **fast unendlich Eng** beisammen!

Frage: Was ist die größte rationale Zahl vor der 7?

⇒ Es gibt sie nicht!

== Eine rationale Zahl hat keinen eindeutig bestimmten Vorgänger oder Nachfolger (nicht so wie bei den ganzen Zahlen).

Dann ist der Zahlenstrahl mit den rationalen Zahlen also voll?

⇒ bei weitem nicht! Die meisten Punkte sind noch frei!
⇒ irrationale Zahlen und überabzählbare Mengen.

<u>Irrationale Zahlen</u>

Was ist die Lösung der Gleichung

$$x^2 = 2?$$

Wäre x eine rationale Zahl, könnten wie sie (gekürzt) als Bruch, also als $x = \frac{a}{b}$ darstellen mit $a, b \in \mathbb{N}$ und teilerfremd.

Dann müsste

gelten
⇒

Wegen dem Faktor 2 in $2 \cdot b^2$ muss der rechte Term $(2 \cdot b^2)$ gerade sein und somit auch der linke Term (a^2).

$\Rightarrow a$ muss gerade sein, da das Quadrat einer ungeraden Zahl wieder eine ungerade Zahl ist

$\Rightarrow$ Wenn a eine gerade Zahl ist, können wir $a = 2 \cdot m$ schreiben mit $m \in \mathbb{N}$ und somit

Sind aber a und b gerade (wie gerade gezeigt), können sie nicht teilerfremd sein!

$\Rightarrow$ der Bruch $\frac{a}{b}$ kann nicht existieren!

Mit diesem Beweis kann man zeigen, daß jede Gleichung $x^2 = c$ in $\mathbb{Q}$ lösbar ist, wenn in der eindeutigen Zerlegung in Primfaktoren von c jeder Primfaktor in gerader Potenz vorkommt.

z.B.:

Dies ist gleichbedeutend mit der Aussage, dass c eine Quadratzahl ist!

Obwohl wir zwischen zwei rationalen Zahlen $a, b \in \mathbb{Q}$ immer eine weitere rationale Zahl finden können die zwischen a und b liegt (z.B. $c = \frac{a+b}{2}$), und daraus folgt, dass die rationalen Zahlen unendlich dicht auf dem Zahlenstrahl beieinander liegen, ist dort immer noch Platz für „andere Zahlen“ die man irrationale Zahlen nennt.

Da wir eine irrationale Zahl nicht als Bruch darstellen können, ist sie eine Dezimalzahl mit einer unendlichen Folge von Nachkommastellen.

Und wie können wir so eine Zahl exakt darstellen?

$\Rightarrow$ gar nicht!

Was wir jedoch machen können ist „gute“ rationale Näherungen zu bestimmen, z.B. mit

dem Babylonischen Wurzelziehen:

z.B. finde eine gute rationale Näherung für $\sqrt{10}$ (Lösung von $x^2 = 10$):

Jedoch ist 10 sicher zu groß, und 1 zu klein

$\Rightarrow$ die „Wahrheit“ ($\sqrt{10}$) muß „irgendwo“ in der Mitte liegen!

$\Rightarrow$ Wir nehmen als nächstes den Mittelwert der beiden Zahlen:

Je öfter wir das wiederholen, desto „besser“ Approximieren wir unsere konstruierte Seitenlänge $\sqrt{10}$!
In einem Algorithmus geht das so:

- ▷ Eingabe x. Starte mit $a_0 = x$, $b_0 = 1$
  $\Rightarrow a_0 \cdot b_0 = x$
- ▷ Berechne $a_1 = \frac{1}{2}(a_0 + b_0)$ und $b_1 = \frac{x}{a_1}$
  $\Rightarrow a_1 \cdot b_1 = x$
- ▷ ...
  $\Rightarrow$

$\Rightarrow$ Wir müssen in jeder Iteration „nur“ a_{n+1} berechnen!

PYTHON: `babylon`

Eine der „irrationalsten“ Zahlen die man kennt ist der **goldene Schnitt**.

Diesen kann man zum Beispiel mit dem Verfahten der inneren Teilung konstruieren:

Was ist das Verhältnis des goldenen Schnitts?

Bezeichnen wir die Länge der grauen Linie mit x, die der roten Linie mit 1, so ist das gesuchte Verhältnis $x : 1$. Die Länge der grünen Linie ist dann $x - 1$ und das Verhältnis von grau zu rot entspricht dem Verhältnis von rot zu grün. Algebraisch bedeutet das, daß

Die positive Lösung dieser Gleichung heißt goldener Schnitt:

Dieses Verhältnis kommt in Natur, Technik, Kunst, ... sehr häufig vor. Es ist jedoch „umstritten“ ob dies wirklich immer „Absicht“ ist oder doch nur Zufall.

Formal approximiert man irrationale Zahlen durch Grenzwerte (siehe zweites Semester).

Für irrationale Zahlen gelten dieselben Rechengesetze wir für die Rationalen Zahlen

Alle Zahlen die wir bisher kennengelernt haben sind **reelle Zahlen** ($\mathbb{R}$)

$\Rightarrow$ Rationale und irrationale Zahlen sind reelle Zahlen

$\Rightarrow$ die irrationalen Zahlen sind die reellen Zahlen ohne die rationalen Zahlen

$$\mathbb{R} \setminus \mathbb{Q}$$

Dies ist eine Mengendifferenz $\Rightarrow$ Details dazu im Teilabschnitt Mengenlehre.

Wir können irrationale Zahlen (fast) immer durch eine rationale Zahl approximieren

$\Rightarrow$ Zwischen zwei „beliebig nahe“ beieinander liegenden rationalen Zahlen ist immer noch Platz für (fast) unendlich viele irrationale Zahlen!

Kurze Einführung in die komplexe Zahlen

Was ist die Lösung von

$$x^2 = -1?$$

$\Rightarrow$ komplexe Zahl $\mathbb{C}$ mit Definition der **imaginären Einheit** $i = \sqrt{-1}$.

Zahl $b \cdot i$ mit $b \in \mathbb{R}$ ist **imaginäre Zahl**.

$a + b \cdot i$ (z.B. $5 + 3 \cdot i$) $a, b \in \mathbb{R}$ ist **komplexe Zahl** wobei a der **Realteil** von $a + b \cdot i$, b der **Imaginärteil** ist.

Menge aller komplexen Zahlen: $\mathbb{C}$

$\Rightarrow i \notin \mathbb{R}$, $i^2 = (\sqrt{-1})^2 = -1, \ldots$

Rechenregeln wie gewohnt:

z.B.:

Lösung von quadratischer Gleichung

$$x^2 + a_1 \cdot x + a_2 = 0$$

ist komplex, falls

z.B. $a_1 = 2$, $a_2 = 2$:

Probe:

Die zu $z = a + b \cdot i$ **konjugiert komplexe Zahl** $\bar{z}$ erhält man durch „umdrehen“ des Vorzeichens des imaginären Teils:

$$\bar{z} = \overline{a + b \cdot i} = a - b \cdot i$$

Komplexe Zahlen kann man sich als Punkte in der komplexen Zahlenebene vorstellen:

Was ist der Abstand von $(3 + 2 \cdot i)$ zum Ursprung?

$\Rightarrow$ Pythagoras!

Dies ist der Betrag und man schreibt:

Mit dem Konjugieren und dem Betrag können wir auch zwei komplexe Zahlen dividieren:

Dies sollte für eine kurze Einführung und für Informatiker vorerst reichen. Für weitere Details (z.B. Polardarstellung) sei auf die Literatur verwiesen.

Literatur

[1] E. Weitz, Konkrete Mathematik (nicht nur) für Informatiker. 2021, https://www.springer.com/de/book/9783662626177