MAGMA WS 18/19

WERNER BLEY

Dieser Kurs richtet sich an Studierende meiner Vorlesung Algebra aus dem WS18/19 und orientiert sich inhaltlich an der Vorlesung. Im allgemeinen Teil dieses Skripts habe ich mich sehr eng an die MAGMA Kurzeinführung von Ulrich Thiel gehalten. Mit seiner freundlichen Genehmigung möchte ich auf sein Skript verweisen, das Sie unter

`http://www.mathematik.uni-stuttgart.de/~thiel/publications/magma_kurzeinfuehrung.pdf`

finden können. Es ist deutlich ausführlicher und geht tiefer auf die Konzepte vom MAGMA ein.

Weiterhin sehr hilfreich ist das MAGMA Handbuch, das Sie unter

`https://magma.maths.usyd.edu.au/magma/handbook`

finden.

1. Grundlagen

1.1. **Die ersten Schritte.** Von der Konsole wird MAGMA mit dem Kommando `magma` gestartet. Man kann MAGMA wie einen Taschenrechner benutzen. Jedes Kommando muss mit einem Semikolon abgeschlossen werden. Probieren Sie

```
2+3;
2*(-3);
(-3)^117;
3/7+2/17;
(1/13)/(-5/19);
```

MAGMA wird mit dem Befehl

```
quit;
```

verlassen.

1.2. **Strukturen von MAGMA.** Jedes Objekt hat in MAGMA einen wohldefinierten *Typ* und gehört einer darüber liegenden Struktur, dem *Parent*, an. Diese Informationen können wir folgt ermittelt werden.

```
> Type(5);
RngIntElt
> Parent(5);
Integer Ring
> Type(3/2);
FldRatElt
> Parent(3/2);
Rational Field
```

Date: 16. Oktober 2024.

Die Ausgaben sind weitgehend selbsterklärend. Die Befehle

```
Z := Integers();
Q := Rationals();
```

erzeugen die ganzen Zahlen bzw. die rationalen Zahlen. Versuchen Sie die folgenden Befehle.

```
a := 9/3;
IsPrime(a);
IsPrime(Z!a);
```

Wie der Name erraten läßt, testet die Funktion `IsPrime`, ob eine ganze Zahl eine Primzahl ist. Die Eingabe für `IsPrime` muss also vom Typ `RngIntElt` sein. Bei der Zuweisung `a := 9/3;` erzeugt aber MAGMA ein Objekt vom Typ `FldRatElt`, daher die Ausgabe *false*, obwohl ja $a = 3$ scheinbar offensichtlich eine Primzahl ist. Will man, daß MAGMA a als Element der ganzen Zahlen, so muss man dies MAGMA mittels dem Typen-Umwandlungsoperator `!` mitteilen. Versuchen Sie

```
Type(a);
Type(Z!a);
Z ! (3/4);
```

1.3. **Intrinsics.** MAGMA hat eine große Anzahl an vordefinierten Funktionen, sogenannten *intrinsics*. Wir haben schon mehrere davon kennen gelernt: `Integers(), Rationals(), IsPrime(n)`. Intrinsics haben eine bestimmte Anzahl von Eingabeobjekten von einem bestimmten Typ. Hier sind weitere Beispiele:

```
Divisors(2020);
PrimeDivisors(2020);
Factorization(2020);
GCD(12, 33);
```

Sowohl die Befehle als auch die Ausgaben sind weitgehend selbsterklärend. Wenn man sich darüber informieren will, wie die verlangte Eingabe ist und wie die Ausgabe zu interpretieren ist, so hilft neben dem MAGMA-Handbuch auch die Eingabe des Intrinsic-Namens abgeschlossen mit Semikolon weiter. Zum Beispiel

```
Divisors;
IsPrime;
```

Sie sehen, daß MAGMA den selben Namen für unterschiedliche Eingabetypen verwendet; MAGMA kann dann anhand der Anzahl und der Typen der Eingabeparameter erkennen, welche Funktion zu verwenden ist.

1.4. **Aussagenlogik.** MAGMA kennt den Typ `BoolElt`, wie wir schon bei der Funktion `IsPrime` gesehen haben. Einfache Beispiele sind

```
a := true;
b := false;
a or b;
a and b;
not a;
```

Sehr oft benötigt man für Vergleichstests `eq` für $=$, `lt` für $<$, `le` für $\leq$, `gt` für $>$ und `ge` für $\geq$.

1.5. **Konditionalausdrücke.** Konditionalausdrücke haben die Gestalt

```
if BoolAusdruck1 then
    Anweisungen 1;
elif BoolAusdsruck 2 then
    Anweisungen 2;
else
    Anweisungen 3;
end if;
```

Dabei kann es beliebig viele `elif`-Teile geben. Betrachten wir das folgende Beispiel. Wir wollen die Funktion $f\colon \mathbb{N}_{>0} \longrightarrow \mathbb{N}_{>0}$ mit

$$f(n) := \begin{cases} \frac{n}{2}, & \text{falls } n \text{ gerade ist,} \\ 3n+1, & \text{sonst.} \end{cases}$$

realisieren. Dazu betrachten wir

```
n := 35;
if IsEven(n) then n := Integers()!(n/2); else n := 3*n+1; end if; n;
```

Man kann einen solchen Ausdruck also auch in eine Zeile schreiben; beim Gebrauch von MAGMA als Taschenrechner hat dies gewisse Vorteile, ist aber eher unleserlich. Wenn wir später eigene Funktionen schreiben, werden wir daher (meist) eine Schreibeweise wie folgt bevorzugen

```
if IsEven(n) then
    n := Integers()!(n/2);
else
    n := 3*n+1;
end if;
```

Da man mit ↑ Befehle zurückholen kann, können Sie die Kurzversion des obigen Konditionalausdrucks nun wieder und wieder durchführen. Was beobachten Sie?

1.6. **Mengen.** Die Verwendung von Mengen in MAGMA ist sehr intuitiv. Wir wollen uns daher hier nur auf das notwendigste beschränken. Versuchen Sie

```
M := {1,2,3,4,5,6,7,8,9};
N := {2..5};
Type(M);
```

Die Kardinalität einer Menge ermittelt man mit `#`, `join` ergibt die Vereinigung und `meet` den Schnitt. Hier einige triviale Beispiele, allesamt selbsterklärend:

```
A := {1,2,3};
B := {3,4,5,6};
#A; #B;
A join B;
A meet B;
A diff B;
4 in A;
4 in B;
4 notin B;
{1,2} subset {1..10};
```

MAGMA erlaubt auch eine Art der Definition von Mengen, die dem Mathematiker intuitiv sofort klar ist. Wir erläutern dieses Konzept an Beispielen:

```
M := {1..20};
X := {a : a in M | IsEven(a)};
Y := {a : a in M | IsOdd(a) and a lt 11};
```

X ist also die Menge der Elemente $a \in M$ mit der Eigenschaft, daß a gerade ist. Y ist die Menge der $a \in M$, die ungerade und kleiner als 11 sind.

Wir wollen die Menge M der Primzahlen p bstimmen, die sich als Summe dreier Primzahlen a, b, c mit $3800 \leq a, b, c \leq 4100$ schreiben lassen. Sodann wollen wir entscheiden, ob es eine Primzahl zwischen $\min(M)$ und $max(M)$ gibt, die nicht in M liegt.

```
M := {a+b+c : a,b,c in {3800..4100} | IsPrime(a) and IsPrime(b) and IsPrime(c)
                                      and IsPrime(a+b+c)};
N := {n : n in {Min(M)..Max(M)} | IsPrime(n)};
N subset M;
M subset N;
N diff M;
```

Aufgabe 1.1. Verstehen Sie diese Zeilen. Warum dauert das relativ lang?

1.7. **Sequenzen.** *Sequenzen* funktionieren ähnlich wie Mengen, allerdings spielt jetzt die Reihenfolge eine Rolle und es sind Wiederholungen möglich. Der Datentyp heißt `SeqEnum`. Hier einige Beispiele:

```
S := [1,2,2,3,3,3,4,4,4,4,5,5,5,5,5];
#S;
S[1]; S[2];
S[1] := 17;
7 in S;
Type(S);
T := [100,101,102];
S cat T;
```

Der Befehl `cat` hängt also zwei Sequenzen aneinander, natürlich unter Beachtung der Reihenfolge.

Die Befehle `SetToSequence` bzw. `SequenceToSet` konvertieren zwischen Mengen und Sequenzen.

```
SetToSequence( {1,2,3,4} );
SequenceToSet( [1,2,2,3,3,3,4,4,4,4] );
```

1.8. **Funktionen und Intrinsics.** In diesem Abschnitt wollen wir lernen, wie man die Funktionalität von MAGMA erweitern kann, indem man eigene Funktionen und Intrinsics schreibt und einbindet.

Die Struktur von Funktionen ist wie folgt:

```
function_name := function(input_1, input_2,....)
    Anweisungen;
    return output_1, output_2, ....;
end function;
```

Hier ein einfaches Beispiel. Wir wollen eine Funktion f definieren, die $x \mapsto x^3$ realisiert.

```
f := function(x)
    return x^3;
end function;
```

```
f(2);
f(1.1);
```

Komplexere Funktionen will man natürlich nicht bei jedem Start von MAGMA neu schreiben. Wir erzeugen daher eine Datei MyFunctions.m (oder Sie wählen sich einen anderen Namen) und schreiben die Funktion in diese Datei. Mit `load ''MyFunctions.m''` kann man dann die Funktionen dieser Datei laden und in MAGMA benutzen.

Wir wollen die Funktion

$$f(n) := \begin{cases} \frac{n}{2}, & \text{falls } n \text{ gerade ist,} \\ 3n+1, & \text{sonst.} \end{cases}$$

als Function realisieren. Editieren Sie dazu die Datei `MyFunctions.m` und speichern Sie sie mit dem Inhalt

```
f := function(n)
    if IsEven(n) then
        n := Integers()!(n/2);
    else
        n := 3*n+1;
    end if;
    return n;
end function;
```

Nach `load ''MyFunctions.m''` können wir nun die Funktion f benutzen. Schleifen werden wir zwar erst kennen lernen, aber intuitiv ist klar, was in den folgenden Zeilen passiert.

```
while n ne 1 do
    n := f(n);
    n;
end while;
```

Für die nächste Aufgabe möchten Sie vielleicht eine `for`-Schleife verwenden. Wir greifen deshalb etwas vor und geben hier die einfachste Form einer solchen Schleife an:

```
for i:=n to m do
    Anweisungen;
end for;
```

Hier sollten n, m ganze Zahlen sein. Zum Beispiel können Sie mit der folgenden Schleife die 20 Primzahlen $> N$ in eine Sequenz `P` anfügen, wobei N eine natürliche Zahl ist.

```
P := [];
N := 2000000000000000000;
for i:=1 to 20 do
    N := NextPrime(N);
    Append(~P, N);
end for;
```

Aufgabe 1.2. Bekanntlich sind die Fibinacci-Zahlen rekursiv definiert durch

$$F_0 := 0, F_1 := 1; F_n := F_{n-1} + F_{n-2}, \text{ falls } n \geq 2.$$

Schreiben Sie eine Funktion `MyFibonacci(n)`, die die n-te Fibonacci-Zahl berechnet. Testen Sie Ihre Implementierung durch Vergleich mit der MAGMA-Funktion `Fibonacci`.

Oftmals ist es nicht sinnvoll, große Objekte, etwa sehr lange Sequenzen, an eine Funktion zu übergeben, da dann eine Kopie des Objekts angelegt wird. Statt dessen kann man auch einen Zeiger auf das Objekt übergeben. In diesem Fall spricht man in MAGMA von `Prozeduren`. Diese liefern keinen Wert zurück, erlauben es aber ein Objekt zu ändern. Prozeduren haben zwei verschieden Arten von Übergabeparametern, nämlich gewöhnliche Übergabeparameter wie in Funktionen sowie Referenzen (Zeiger) auf Objekte. Eine Referenz auf ein Objekt X übergibt man mittel $\sim X$. Hier ein Beispiel.

```
MyCat := procedure(~seq, n)
    seq := seq cat [n];
end procedure;

S := [1..10^5];
#S;
S[#S];
MyCat(~S, -1);
#S;
S[#S];
```

Wir kommen nun zu Intrinsics. Intrinsics sind Funtionen oder Prozeduren, die MAGMA so behandelt, als wären es feste Bestandteile von MAGMA. Daher verlangt die Implementierung auch etwas mehr Sorgfalt; MAGMA will genauere Informationen über die Funktion bzw. die Prozedur. Eine Intrinsic wird wie folgt defieniert

```
intrinsic Intrinsic_Name(Eingabe_1 :: Typ_1, ..., Eingabe_n :: Typ_n) -> AusgabeTyp_1, ..., A
{Beschreibung der Intrinsic.}
    Anweisungen;
    return Ausgabe_1, ..., Ausgabe_m;
end intrinsic;
```

Hier ein Beispiel. Wir realisieren die Fibonacci-Zahlen als Intrinsic.

```
intrinsic MyFibo(n :: RngIntElt) -> RngIntElt
{Meine Fibonaccizahlen.}
   if n eq 0 then return 0; end if;
   if n eq 1 then return 1; end if;
   a := 0; b := 1;
   for i:=2 to n do
       c := b;
       b := a+b;
       a := c;
   end for;
   return b;
end intrinsic;
```

Schreiben Sie diese Intrinsic in eine Datei *MyIntrinsics.m* und geben Sie in MAGMA den Befehl `Attach('MyIntrinsics.m')` ein. Damit wird die Funktion eingebunden. Versuchen Sie nun

```
MyFibo;
```

Eine Intrinsic akzeptiert auch Referenzen als Eingabeparameter, allerdings darf dann in der Intrinsic kein return-Statement enthalten sein. Eine MAGMA-eigene Intrinsic, wo wir dies oft verwenden werden, ist `Append(~ S, x)`. Hier wird an eine Liste S bestehend aus Objekten eines gewissen Typs eine weiteres Objekt x des gleichen Typs T angefügt. Ein Beispiel

```
LangeListe := [1..10^7];
#LangeListe;
Append(~LangeListe, -1);
#LangeListe;
LangeListe[ #LangeListe ];
```

1.9. **Schleifen.** Wir wollen uns hier auf `for`- und `while`-Schleifen beschränken und diese anhand von Beispielen kennenlernen. Zunächst zur `for`-Schleife.

```
for i:=0 to 99 do
    MyFibonacci(i) eq Fibonacci(i);
end for;
```

vergleicht Ihre Implementierung mit der MAGMA-eigenen Implementierung anhand der ersten hundert Fibonacci-Zahlen. Wenn Ihre Funktion richtig arbeitet, so bekommen Sie 100 mal den wahrheitswert `true`. Äquivalent dazu sind

```
for i in [0..99] do
    MyFibonacci(i) eq Fibonacci(i);
end for;
```

oder

```
for i in {0..99} do
    MyFibonacci(i) eq Fibonacci(i);
end for;
```

Auch `for`-Schleifen der folgenden Art sind erlaubt

```
for i,j in {1,2,3} do
    print "i = ", i, " j = ", j;
end for;
```

Hier haben wir eine etwas primitive Art der Ausgabe benutzt, die weitgehend selbsterklärend ist. Wenn nicht, so konsultieren Sie bitte das Handbuch.

Die Form einer `while`-Schleife ist wie folgt

```
while BoolElt do
    Anweisungen;
end while;
```

Wir betrachten wieder einige Beispiele. Wir wollen eine Liste der Primzahlen $\leq N$ erzeugen.

```
N := 200;
p := 2;
P := [];
while p le N do
    Append(~P, p);
    p := NextPrime(p);
end while;
```

Die folgende Schleife sucht die erste Primzahl $\geq N$, die kongruent zu a modulo m ist. Dabei sollte $ggT(a, m) = 1$ gelten.

```
N := 500;
a := 1;
m := 133;
p := NextPrime(N-1);
found := p mod m eq a;
while not found do
    p := NextPrime(p);
    found := p mod m eq a;
end while;
p;
```

Wir wollen aus den obigen Beispielen Intrinsics machen und diese dann mit `Attach('MyIntrinsics.m')` einbinden.

```
intrinsic MyPrimes(N :: RngIntElt) -> SeqEnum
{Sequenz der Primzahlen kleiner oder gleich N.}
    p := 2;
    P := [];
    while p le N do
        Append(~P, p);
        p := NextPrime(p);
    end while;
   return P;
end intrinsic;
```

Aufgabe 1.3. Beweisen Sie, dass die Bedingung $ggT(m, a) = 1$ notwendig ist. Das die Bedingung hinreichend für die Existenz einer Primzahl p mit $p \equiv a \pmod{m}$ ist, ist nicht-trivial. Tatsächlich gibt es unendlich viele solcher Primzahlen. Schreiben Sie ein

`intrinsic MyPrime(N :: RngIntElt, a :: RngIntElt, m :: RngIntElt) -> RngIntElt`,

das die kleinste Primzahl $p \geq N$ mit $p \equiv a \pmod{m}$ liefert.

2. Gruppentheorie, Teil 1

Wir wollen in diesem Abschnitt verschiedene Typen von Gruppen kennen lernen und verstehen, wie sie in MAGMA dargestellt werden und wie man mit ihnen umgehen kann.

Wir beginnen mit den endlichen abelschen Gruppen. Wie in der Vorlesung bezeichne C_n die zyklische Gruppe der Ordnung n. Hierzu beachte man, daß je zwei zyklische Gruppen der Ordnung n isomorph sind, so daß die Notation gerechtfertigt ist. In MAGMA wird die C_n durch den Befehl `CyclicGroup(GrpAb, n)` erzeugt. Auch der Aufruf `CyclicGroup(n)` ist korrekt, liefert aber keine Gruppe vom Typ `GrpAb`, sondern eine Gruppe vom Typ `GrpPerm`. Hierzu kommen wir später; zunächst wollen wir den Typ `GrpAb` genauer betrachten. Geben Sie

```
C := CyclicGroup(GrpAb, 14); C;
```

ein. MAGMA liefert uns eine abelsche Gruppe isomorph zu $\mathbb{Z}/14\mathbb{Z}$ zurück. Der Erzeuger kann mit `C.1` angesprochen werden. Schöner ist es unter Umständen, wenn man

```
C<a> := CyclicGroup(GrpAb, 14); C;
```

benutzt. Dann kann man auf den Erzeuger mittels `a` zugreifen. Versuchen sie die folgenden Eingaben, überlegen Sie sich aber bereits zuvor das Ergebnis.

```
Order(a);
Order(2*a);
Order(7*a);
8*a eq 22*a;
Order(3*a) eq Order(a);
{x : x in C} eq {i*a : i in [5..18]};
```

Wir betrachten nun abelsche Gruppen der Form $C_{n_1} \times \ldots \times C_{n_l}$ mit $l, n_1, \ldots, n_l \in \mathbb{N}$. Dies ist eine abelsche Gruppe der Ordnung $n_1 \cdots n_l$. Der Aufruf

```
AbelianGroup([n_1,..., n_l])
```

erzeugt diese Gruppe. Wie wir in der Vorlesung lernen werden, ist jede endliche abelsche Gruppe bis auf Isomorphie von dieser Form. Mit dem Kommando

```
A := AbelianGroup([2,3,5]);
```

erzeugen wir also eine abelsche Gruppe, die isomorph zu $\mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/3\mathbb{Z} \times \mathbb{Z}/5\mathbb{Z}$ ist. Die Erzeuger heißen `A.1, A.2` und `A.3`. Versuchen sie die folgenden Eingaben, überlegen Sie sich aber wieder vorab das Ergebnis.

```
[Order(A.1), Order(A.2), Order(A.3)];
A<a,b,c> := AbelianGroup([2,3,5]);
a eq A.1; b eq A.2; c eq A.3;
Order(a+b);
a+b eq a + 4*b;
Order(A);
```

Der Aufruf `AbelianGroup([n])` erzeugt ebensfalls die zyklische Gruppe C_n. Hierzu folgendes Beispiel:

```
A<a> := AbelianGroup([100]);
B<b> := CyclicGroup(GrpAb, 100);
bool, f := IsIsomorphic(A,B);
bool; f;
f(a);
```

Sicherlich haben Sie sich gewundert, warum `CyclicGroup` zwei Argumente hat. Wie bereits weiter oben erwähnt, ist auch `CyclicGroup(n)` korrekt, liefert aber eine Gruppe vom Typ `GrpPerm`. Versuchen Sie folgende Eingaben:

```
C<c> := CyclicGroup(100);
C;
Type(C);
IsIsomorphic(A,C);
```

MAGMA weigert sich also Gruppen von verschiedenem Typ auf Isomorphie zu testen. Mit dem Befehl `PermutationGroup(A)` kann man aus einer Gruppe A vom Typ `GrpAb` eine Gruppe vom Typ `GrpPerm` machen. Hier ein Beispiel

```
C<c> := CyclicGroup(100);
C;
Type(C);
IsIsomorphic(PermutationGroup(A), C);
```

`GrpPerm` steht für endliche Permutationsgruppen. Dies ist der Typ von Gruppen, für die sehr viele Probleme algorithmische Lösungen haben. Natürlich kann man diese Probleme auch für endliche abelsche Gruppen lösen. Hier sind die Probleme jedoch meist trivial oder haben eine einfache Lösung.

Permutationsgruppen sind Untergruppen einer symmetrischen Gruppe S_n. Ohne Einschränkung stellen wir uns die S_n meist als die Gruppe der Permutationen der Menge $\{1, \ldots, n\}$ vor. Wie wir in der in der Vorlesung sehen werden, kann man jede endliche Gruppe G als Permutationsgruppe darstellen (mit einem geeigneten n). Hier ist die Beweisidee, die wir später an Beispielen umsetzen wollen. Sei $\sigma \in G$. Dann defieren wir

$$f_\sigma \colon G \longrightarrow G, \quad \tau \to \sigma\tau.$$

Die Abbildung

$$G \longrightarrow S(G), \quad \sigma \to f_\sigma$$

ist eine Einbettung (d.h. ein injektiver Gruppenhomomorphismus) und wir könne G mit seinem Bild identifizieren. Hierbei bezeichnet $S(G)$ die Gruppe der Permutationen der Menge G. Numerieren wir die Gruppenelemente, so können wir natürlich $S(G)$ und $S_{|G|}$ identifizieren.

Wir müssen also zunächst die symmetrischen Gruppen S_n untersuchen. Sie werden in MAGMA durch `SymmetricGroup(n)` oder kurz durch `Sym(n)` erzeugt. Versuchen Sie

```
S := Sym(3);
Set(S);
```

Wie wir sehen, verwendet MAGMA für die Elemente der symmetrischen Gruppen die übliche Zyklenschreibweise. Im Gegensatz zur Vorlesung wird in MAGMA jedoch $\sigma\tau$ nicht wie "σ nach τ" interpretiert, sondern wie "τ nach σ". Grund hierfür ist die MAGMA-Konvention, daß Abbildungen stets von rechts wirken. Es gilt also zum Beispiel $(1,2)(2,3) = (1,3,2)$. Die entsprechende Eingabe in MAGMA ist

```
s := S ! (1,2)(2,3);
s;
```

Betrachten wir nun das folgende Beispiel.

```
C<a> := CyclicGroup(4); C;
C ! (1,2)(3,4);
C ! (1,3)(2,4);
```

C wird also dargestellt als Untergruppe der S_4 erzeugt von $a = (1,2,3,4)$. Da

$$\langle a \rangle = \{id, (1,2,3,4), (1,3)(2,4), (1,4,3,2)\}$$

erklären sich die obigen Ausgaben von MAGMA.

Weitere wichtige Gruppen, die in MAGMA als Permutationsgruppen dargestellt werden, sind die Diedergruppen. Das intrinsic `DihedralGroup(n)` erzeugt die Diedergruppe D_n mit $|D_n| = 2n$. Mit dem Befehl `AlternatingGroup(n)` erzeugt man die alternierende Guppe A_n.

Aufgabe 2.1. Testen Sie mit MAGMA, ob $D_3 \simeq S_3$ und $C_3 \simeq A_3$. Geben Sie gegebenenfalls einen expliziten Isomorphismus an.

Als letzten Typ von Gruppen wollen wir Matrixgruppen kennen lernen. Wir beschränken uns hier auf die Konstruktion einer $\mathrm{Gl}_n(\mathbb{F}_p)$ und ihrer Untergruppen. Hierbei bezeichnet $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ den Körper mit p Elementen. Ferner werden wir

die Quaternionengruppe der Ordnung 8 zunächst als Matrixgruppe, und darauf aufbauend als Permutationsgruppe realisieren.

Wir betrachten folgende Sequenz.

```
QX<x> := PolynomialRing( Rationals());
F<i> := NumberField(x^2+1);
M := GL(2,F);
E := M![1,0,0,1];
I := M![i,0,0,-i];
J := M![0,-1,1,0];
K := M![0,-i,-i,0];
Q := MatrixGroup<2,F|[I,J,K]>;
```

Zunächst definieren wir den Polynomring $\mathbb{Q}[x]$ über den rationalen Zahlen. Der zweite Befehl erzeugt die Gaußschen Zahlen $\mathbb{Q}(i)$. `M` ist die Gruppe der invertierbaren 2×2-Matrizen mit Einträgen in $F = \mathbb{Q}(i)$. Durch die folgenden Zeilen werden die Matrizen

$$E = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, I = \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}, J = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, K = \begin{pmatrix} 0 & -i \\ -i & 0 \end{pmatrix}$$

definiert, und schließlich in der letzten Zeile die Untergruppe der $\mathrm{Gl}_2(F)$, die durch I, J, K erzeugt wird.

Aufgabe 2.2. Überprüfen Sie mit Magma, ob Q mit der in Aufgabe 2 des 2.Übungsblatts definierten Quaternionengruppe

$$\left\langle \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix} \right\rangle$$

übereinstimmt.

Die Befehle

```
IsAbelian(Q);
#Q;
```

zeigen, dass Q tatsächlich nicht-abelsch von der Ordnung 8 ist. Aus den Übungen wissen wir außerdem, daß es genau zwei nicht-abelsche Gruppen der Ordnung 8 (bis auf Isomorphie) gibt. Versuchen Sie

```
D := DihedralGroup(4);
IsIsomorphic(Q, D);
```

Wie zu erwarten war, weigert sich MAGMA, da `Q` vom Typ `GrpMat` ist und nicht wie `D` vom Typ `GrpPerm`. Wir wollen daher die Quaternionengruppe auch noch als Permutationsgruppe realisieren und verwenden dazu die obige Beweisskizze. Betrachten Sie

```
S8 := Sym(8);
Qelts := [g : g in Q];
v := [Index(Qelts, I*g) : g in Qelts]; v;
sigma := S8!v;
w := [Index(Qelts, J*g) : g in Qelts];
tau := S8!w;
H := sub<S8 | [sigma, tau]>;
#H;
D := DihedralGroup(4);
```

```
#D;
IsIsomorphic(D, H);
IsAbelian(H);
IsAbelian(D);
#D eq #H;
```

Aufgabe 2.3. Versuchen Sie die obige Befehlssequenz zu verstehen. Stellen Sie dazu den Zusammenhang zur obigen Beweisskizze her.

Wir wollen nun eine Funktion schreiben, die die D_4 als Permutationsgruppe erzeugt. Dazu stellen wir uns die D_4 als die Gruppe der Isometrien eines Quadrates vor. Wir numerieren die Ecken von 1 bis 4. Jede Isometrie ist durch die zugehörige Permutation der Ecken eindeutig bestimmt. Also können wir die D_4 als Untergruppe der S_4 darstellen. Aus der Vorlesung wissen wir, daß die D_4 durch die Drehung σ um 90 Grad und eine Spiegelung τ erzeugt wird. Dies sollte ausreichen, um die folgende Funktion zu verstehen.

```
MyD4 := function()
    S4 := Sym(4);
    sigma := S4 ! (1,2,3,4);
    tau := S4 ! (1,2)(3,4);
    return sub<S4 | [sigma, tau]>;
end function;
```

Wir testen die Funktion durch

```
G := MyD4();
G;
IsIsomorphic(G, DihedralGroup(4));
```

Aufgabe 2.4. Schreiben Sie Funktionen MyD5() und MyD6(), und allgemeiner eine Funktion MyDn(n).

Aufgabe 2.5. Schreiben Sie eine Funktion MyQ8(), die die Quaternionengruppe der Ordnung 8 als Permutationsgruppe erzeugt.

Aufgabe 2.6. Realisieren Sie MyDn(n) und MyQ8() ebenfalls als intrinsics und speichern Sie die intrinsics in MyIntrinsiscs.m ab. In den folgenden Sitzungen wollen wir diese Datei stets zur Verfügung haben.

3. Der Chinesische Restsatz (für den Ring $\mathbb{Z}$)

Der Chinesische Restsatz für beliebige kommutative Ringe R ist mittlerweile aus der Algebra bekannt. Wir werden den Satz hier nochmals für den Spezialfall $R = \mathbb{Z}$ formulieren und beweisen. Der Beweis ist, wie wir sehen werden, konstruktiv und wir können ihn fast wortwörtlich in eine MAGMA-Funktion umsetzen.

Theorem 3.1. *Sei $n \geq 1$ eine natürliche Zahl und $a_1, \ldots, a_n \in \mathbb{Z}$. Seien $m_1, \ldots, m_n \in \mathbb{N}$ paarweise teilerfremde, natürliche Zahlen. Dann gibt es eine ganze Zahl $x \in \mathbb{Z}$ mit*

$$x \equiv a_i \pmod{m_i}, \quad i = 1, \ldots, n.$$

Die Zahl x ist modulo $m_1 \cdots m_n$ eindeutig bestimmt, d.h., falls $y \in \mathbb{Z}$ ebenfalls die simulatanen Kongruenzen

$$y \equiv a_i \pmod{m_i}, \quad i = 1, \ldots, n,$$

erfüllt, so gilt $x \equiv y \pmod{(m_1 \cdots m_n)}$.

Proof. Dem Beweis schicken wir die folgende Tatsache voraus.

Tatsache: Falls $a, b \in \mathbb{Z}$ und $d := \mathrm{ggT}(a, b)$, so gibt es $p, q \in \mathbb{Z}$ mit $pa + qb = d$.

Den Beweis hierzu werden wir (in größerer Allgemeinheit, nämlich für sogenannte Hauptidealringe) in der Vorlesung zur Zahlentheorie führen. Für $R = \mathbb{Z}$ haben wir den Beweis mittels erweitertem euklidischen Algorithmus in der Zahlentheorie geführt.

Nun zum Beweis des Chinesischen Restsatzes. Der Beweis erfolgt mittels Induktion über n. Falls $n = 1$, so setze $x := a_1$. Da wir den Fall $n = 2$ im Induktionsschritt brauchen werden, zeigen wir die Behauptung zunächst in diesem Fall. Da m_1 und m_2 nach Voraussetzung teilerfremd sind, gibt es $\tilde{p}, \tilde{q} \in \mathbb{Z}$ mit $1 = \tilde{p}m_1 + \tilde{q}m_2$. Multiplikation mit $a_1 - a_2$ liefert

$$a_1 - a_2 = pm_1 + qm_2 \text{ mit } p = (a_1 - a_2)\tilde{p}, q = (a_1 - a_2)\tilde{q}.$$

Für $x := a_1 - pm_1 = a_2 + qm_2$ gilt wie gefordert

$$x \equiv a_1 \pmod{m_1}, \quad x \equiv a_2 \pmod{m_2}.$$

Wir kommen nun zum Induktionsschritt $n - 1 \longrightarrow n$. Dazu setzen wir $M := m_1 \cdots m_{n-1}$. Nach Induktion gibt es eine Lösung $y \in \mathbb{Z}$ von

$$y \equiv a_i \pmod{m_i}, \quad i = 1, \ldots, n-1.$$

Da $\mathrm{ggT}(M, m_n) = 1$, können wir wie im Fall $n = 2$ die simultanen Kongruenzen

$$x \equiv y \pmod{M}, \quad x \equiv a_n \pmod{m_n}$$

lösen. Da $m_i \mid M$ folgt $x \equiv y \equiv a_i \pmod{m_i}$ für $i = 1, \ldots, n-1$.

Zur Eindeutigkeit: Es gelte sowohl

$$x \equiv a_i \pmod{m_i}, \quad i = 1, \ldots, n$$

als auch

$$y \equiv a_i \pmod{m_i}, \quad i = 1, \ldots, n.$$

Dann folgt $x \equiv y \pmod{m_i}, i = 1, \ldots, n$, was äquivalent zu $m_i \mid (x-y), i = 1, \ldots, n$, ist. Da die m_i paarweise teilerfremd sind, folgt

$$m_1 \cdots m_n \mid (x - y),$$

beziehungsweise äquivalent hierzu

$$x \equiv y \pmod{m_1 \cdots m_n}.$$

□

Wir wollen diesen Beweis nun umsetzen in eine MAGMA-Funktion. Dazu nehmen wir die intrinsic XGCD zur Hilfe. Der Aufruf *d,p,q := XGCD(a,b)* liefert die Zahlen d, p, q wie in der obigen Tatsache.

Der Input für die Funktion seien zwei Listen $a := [a_1, \ldots, a_n], m := [m_1, \ldots, m_n]$. Wir verzichten auf jegliche Konsistenzüberprüngen wie etwa *#a eq #m*, da es uns hier nur auf die Implementierung der wesentlichen Teile des Algorithmus ankommt.

```
MyCR := function(a, m)
    M := 1;
    x := a[1];
    for i:=2 to #a do
        y := x;
        M := M * m[i-1];
```

```
        d,p,q := XGCD(M, m[i]);
        if d ne 1 then
            print "FEHLER: die m_i sind nicht paarweise teilerfremd.";
            return -1;
        end if;
        p := (y - a[i])*p;
        q := (y - a[i])*q;
        x := y - p*M;
    end for;
    return x mod &*m ;
end function;
```

Aufgabe 3.2. Bevor Sie die Funktion implementieren, spielen Sie bitte selbst Algorithmus und führen die Funktion auf dem Papier am Beispiel

$$\begin{aligned} x &\equiv 1 \pmod 3, \\ x &\equiv 2 \pmod 5, \\ x &\equiv 3 \pmod 7 \end{aligned}$$

durch. Vergleichen Sie Beweis und Implementierung.

Aufgabe 3.3. Überprüfen Sie die Sinnhaftigkeit der Fehlermeldung. Zeigen Sie dazu: Die m_i sind genau dann paarweise teilerfremd, wenn für alle $i \in \{2, \dots, n\}$ gilt: $\mathrm{ggT}(m_1 \cdots m_{i-1}, m_i) = 1$.

Wie wir in der Vorlesung lernen werden, kann man den Chinesischen Restsatz völlig analog in beliebigen sogenannten Euklidischen Ringen formulieren. Der Beweis ist praktisch identisch. Unsere Standardbeispiele für Eukldische Ringe sind $\mathbb{Z}$ und Polynomringe über einem Körper. Da MAGMA in Funktionen keine Typenüberprüfung vornimmt, können wir die Funktion MyCR auch auf Polynome anwenden. Versuchen Sie die folgenden Eingaben.

```
QX<x> := PolynomialRing(Rationals());
a := [x-1,2*x,1,2,3];
m := [x^2 - 5, x^3 - 1, x^3, x^17 - x, x^15 -x^10 - 7];
f := MyCR(a,m);
// Fehler, da die m_i nicht paarweise teilerfremd sind.


a := [x-1,2*x,1,2,3];
m := [x^2 - 5, x^3 - 1, x^3, x^17 - x+1, x^15 -x^10 - 7];
f := MyCR(a,m);
f;
// Kontrolle
[f mod m[i] eq a[i] mod m[i] : i in [1..#a]];
```

4. Einheiten in $\mathbb{Z}/m\mathbb{Z}$

Der (erweiterte) Euklidische Algorithmus spielt in der algorithmischen Zahlentheorie eine bedeutende Rolle. Auch im folgenden Satz ist er das wesentliche Hilfsmittel im Beweis, der sich wieder eins zu eins in eine MAGMA-Funktion umsetzen läßt. Genaueres hierzu nach dem Satz und Beweis.

Theorem 4.1. *Sei $m \geq 2$ eine natürliche Zahl und $a \in \mathbb{Z}$. Dann gilt:*

$$\overline{a} \in (\mathbb{Z}/m\mathbb{Z})^\times \iff \mathrm{ggT}(a,m) = 1.$$

Proof. Falls a modulo m invertierbar ist, so gibt es ein $b \in \mathbb{Z}$ mit $ab \equiv 1 \pmod{m}$. Also gibt es ein $v \in \mathbb{Z}$ mit $ab - 1 = vm$, oder äquivalent dazu, $ab - vm = 1$. Wäre nun $d := \mathrm{ggT}(a,m) > 1$, so wäre $d > 1$ auch ein Teiler von 1, was absurd ist. Sei umgekehrt $\mathrm{ggT}(a,m) = 1$. Dann gibt es $p, q \in \mathbb{Z}$, so dass $1 = pa + qm$. Lesen wir diese Gleichung modulo m, so erhalten wir $\overline{p} \cdot \overline{a} = \overline{1}$, d.h. $\overline{p}$ ist das Inverse von $\overline{a}$. □

Aufgabe 4.2. Schreiben Sie eine intrinsic MyIsInvertibe(a,m), die zu a und m entscheidet, ob a modulo m invertierbar ist und gegebenenfalls das Inverse $\overline{b}$ berechnet. Die Ausgabe soll also aus zwei Werten bestehen, einem BoolElt und einem RngIntElt. Geben Sie für $\overline{b}$ einen Vertreter $b \in \mathbb{Z}$ mit $1 \leq b \leq m-1$ zurück.

Der obige Satz liefert das folgende Korollar.

Corollary 4.3. *Sei $m \geq 2$ eine natürliche Zahl. Dann gilt:*

$$\mathbb{Z}/m\mathbb{Z} \text{ ist ein Körper} \iff m \text{ ist eine Primzahl.}$$

Aufgabe 4.4. Führen Sie die folgende Schleife aus:

```
m := 2;
for m:=2 to 20 do
    print "m = ", m, "    ", [<a, MyIsInvertible(a,m)> : a in [1..m-1]];
end for;
```

Beweisen Sie nun das Korollar.

5. Zyklische Untergruppen, Untergruppen, Normalteiler

In diesem Abschnitt wollen wir uns mit Untergruppen einer gegegebenen Gruppe G beschäftigen. Mit dem Befehl `sub< G | [`$g_1, \ldots, g_n$`] >` kann man die von den Elementen $g_1, \ldots, g_n \in G$ erzeugte Untergruppe erzeugen. Geben Sie folgende Zeilen ein.

```
G := Sym(5);
Gens := SetToSequence( Generators(G) );
tau := Gens[1];
sigma := Gens[2];
U1 := sub<G | [tau]>;
U2 := sub<G | [sigma]>;
U := sub<G | [sigma, tau]>;
```

U_1 und U_2 sind also jeweils von einem Element erzeugt, d.h. dies sind die von τ bzw. σ erzeugten zyklischen Untergruppen von G. Da für eine zyklische Untergruppe $\langle\gamma\rangle \leq G$ ganz allgemein (unabhängig von unserer speziellen Situation) mit einem $\gamma \in G$ von endlicher Ordnung gilt, daß

$$\langle\gamma\rangle = \{\gamma, \gamma^2, \ldots, \gamma^{\mathrm{ord}(\gamma)} = 1\},$$

ist in unserem Beispiel $|U_1| = \mathrm{ord}(\tau) = 2$ und $|U_2| = \mathrm{ord}(\sigma) = 5$. Da ferner G von σ und τ erzeugt wird, gilt $U = G$. Wir überprüfen dies mittels

```
Order(tau) eq #U1;
Order(sigma) eq #U2;
U eq G;
```

Wir wollen nun selbst eine einfache Funktion schreiben, die die Ordnung eines Elements $g \in G$ für eine endliche Gruppe G bestimmt.

```
Ordnung := function(g)
    G := Parent(g);
    a := g;
    o := 1;
    while a ne One(G) do
        o := o + 1;
        a := a*g;
    end while;
    return o;
end function;
```

Hier zwei Erläuterungen: Jedes Gruppenelement 'lebt' in einer Gruppe; diese erhält man durch den Befehl `Parent(g)`. Mit `One(G)` haben wir Zugriff auf das neutrale Element von G.

Wir wollen nun in $G := \mathrm{Gl}(2, \mathbb{F}_q)$, $q = p^n$, die fogenden zwei Untergruppen

$$\begin{aligned} D &= \left\{ \begin{pmatrix} \alpha & 0 \\ 0 & \beta \end{pmatrix} \mid \alpha, \beta \in \mathbb{F}_q^\times \right\}, \\ B &= \left\{ \begin{pmatrix} 1 & \alpha \\ 0 & 1 \end{pmatrix} \mid \alpha \in \mathbb{F}_q \right\}, \end{aligned}$$

als Untergruppen der $\mathrm{Gl}(2, \mathbb{F}_q)$ realisieren. Dazu die folgenden Vorbemerkungen. Den endlichen Körper mit q Elementen erzeugt man mit dem Befehl `GaloisField(q)`, oder kurz mit `GF(q)`. In der Vorlesung werden wir zeigen, daß endliche Untergruppen der multiplikativen Gruppe eines Körpers stets zyklisch sind. Die folgende Sequenz berechnet ein Element $w \in \mathbb{F}_q^\times$ mit $\langle w \rangle = \mathbb{F}_q^\times$ für $q = 11$.

```
q := 11;
F := GF(q);
U, j := UnitGroup(F); U; j;
w := j(U.1);
```

Durch die folgenden Befehle

```
G := GL(2, F);
A := G ! [w,0,0,1];
H := sub<G | [A]>;
```

wird zunächst die Matrix $A = \begin{pmatrix} w & 0 \\ 0 & 1 \end{pmatrix}$ und sodann die Untergruppe

$$H = \left\{ \begin{pmatrix} \alpha & 0 \\ 0 & 1 \end{pmatrix} \mid \alpha \in \mathbb{F}_q^\times \right\}$$

erzeugt.

Aufgabe 5.1. Schreiben Sie eine intrinsic MyPrimRoot(F :: FldFin) -> FldFinElt, die eine Primitivwurzel w zurückliefert.

Aufgabe 5.2. Schreiben Sie eine intrinsic

`intrinsic ComputeGDB(q :: RngIntElt) -> GrpMat, GrpMat, GrpMat`

die die Gruppen G, D, B zurück liefert. Bestimmen Sie Formeln für die Ordnungen von G, D und B.

Wir wollen nun zu einer Untergruppe U von G und $g \in G$ die Linksnebenklasse gU sowie die Rechtsnebenklasse Ug bestimmen. Durch die Verwendung von intrinsics wird, wie wir gleich sehen werden, der Aufruf sehr intuitiv.

```
intrinsic Nebenklasse(g :: GrpPermElt, U :: GrpPerm) -> SetEnum
{Berechnet die Linksnebenklasse gU.}
    return {g*u : u in U};
end intrinsic;

intrinsic Nebenklasse(U :: GrpPerm, g :: GrpPermElt) -> SetEnum
{Berechnet die Linksnebenklasse gU.}
    return {u*g : u in U};
end intrinsic;
```

Testen Sie die Funktion, etwa mit folgendem Beispiel

```
S := Sym(3);
sigma := S.1;
U := sub<S | [S!(1,2)]>;
sigmaU := Nebenklasse(sigma, U);
Usigma := Nebenklasse(U, sigma);
```

Nachdem wir diese Funktionen zur Verfügung haben, wollen wir sie verwenden, um die Menge der Linksnebenklassen G/U sowie die Menge der Rechtsnebenklassen $U\backslash G$ berechnet.

```
intrinsic Linksnebenklassen(G :: GrpPerm, U :: GrpPerm) -> SeqEnum, SeqEnum
{Berechnet G/U.}
    Gset := Set(G);
    R := [Id(G)];
    Q := Nebenklasse(Id(G), U);
    Nk := [ Q ];
    while #Q lt #G do
        g := [g : g in Gset diff Q][1];
        gU := Nebenklasse(g, U);
        Q := Q join gU;
        Append(~R, g);
        Append(~Nk, gU);
    end while;
    return Nk, R;
end intrinsic;

intrinsic Rechtsnebenklassen(G :: GrpPerm, U :: GrpPerm) -> SeqEnum, SeqEnum
{Berechnet U\G.}
    Gset := Set(G);
    R := [Id(G)];
    Q := Nebenklasse(U, Id(G));
    Nk := [ Q ];
    while #Q lt #G do
        g := [g : g in Gset diff Q][1];
        Ug := Nebenklasse(U, g);
        Q := Q join Ug;
```

```
        Append(~R, g);
        Append(~Nk, Ug);
    end while;
    return Nk, R;
end intrinsic;
```

Aufgabe 5.3. Spielen Sie die Funktionen am obigen Beispiel durch.

Nachdem Sie sich klar gemacht haben, wie die Funktionen arbeiten, versuchen wir die folgenden Beispiele.

```
G := Sym(3);
U := sub<G | [G!(1,2)]>;
L := Linksnebenklassen(G, U);
R := Rechtsnebenklassen(G, U);
L eq R;

H := AlternatingGroup(4);
V := sub<H | [H!(1,2)(3,4), H!(1,3)(2,4)]>;
L := Linksnebenklassen(H, V);
R := Rechtsnebenklassen(H, V);
L eq R;
```

Wie das erste Beispiel zeigt, stimmen im Allgemeinen Rechts- und Linksnebenklassen nicht überein. Wie wir in der Vorlesung sehen, ist dies aber die Voraussetzung dafür, daß die von G auf G/U induzierte Multiplikation wohldefiniert ist. Für zwei Nebenklassen g_1U, g_2U möchte man

$$g_1U \cdot g_2U := g_1g_2U$$

definieren. Dies soll unabhängig von der Wahl von Vertretern g_1 bzw. g_2 sein.

Aufgabe 5.4. Finden Sie im ersten Beispiel oben zwei Nebenklassen g_1U und g_2U, so dass die so definierte Multiplikation nicht wohldefiniert ist.

Wir erinnern an die folgende Definition.

Definition 5.5. Sei G eine Gruppe und N eine Untergruppe von G. Dann ist N ein Normalteiler, falls für alle $g \in G$ gilt: $gU = Ug$.

Offensichtlich ist die definierende Bedingung äquivalent zu $g^{-1}Ug = U$ für alle $g \in G$.

Aufgabe 5.6. Zeigen Sie, daß es reicht, die Normalteilereigenschaft für die Erzeugenden von G und U zu überprüfen. Schreiben Sie eine intrinsic IstNormal (G :: GrpPerm, U :: GrpPerm) -> BoolElt.

Aufgabe 5.7. Schreiben Sie eine intrinsic IstZentral(g :: GrpElt) -> BoolElt.

Aufgabe 5.8. Bestimmen Sie die sämtlichen Untergruppen der D_4, Q_8 und der A_4. Zeichnen Sie jeweils ein Untergruppendiagramm. Welche der Untergruppen sind jeweils Normalteiler.

6. Ringe und Quotienten

In diesem Abschnitt wollen wir uns mit kommutativen Ringen und ihren Quotienten beschäftigen. Hier ein einfaches Beispiel.

```
R := Integers();
m := 17;
I := ideal<R | m>;
Q, q := quo<R | I>;
```

Hierdurch erzeugen wir den Quotientenring $\mathbb{Z}/m\mathbb{Z}$. Durch den Befehl `ideal<Ring | a_1, ..., a_n>` erzeugt man das von den Elementen $a_1, \ldots, a_n \in R$ erzeugte Ideal von R, und durch `quo<R | I>` erhält man den Quotientenring $Q = R/I$ sowie die kanonische Abbildung $R \longrightarrow R/I$, die wir hier q getauft haben.

Betrachten Sie folgende Sequenz von Befehlen. Wie immer sollten Sie sich vorher die Ergebnisse überlegen.

```
J := ideal<R | 34, 51>;
I eq J;
P, p := quo<R | J>;
P eq Q;
```

Aus der Vorlesung wissen wir, dass R/I genau dann ein Körper ist, wenn I maximal ist. Wenn nun R ein Hauptidealring (kurz, HIR) ist, so ist I genau dann maximal, wenn I prim ist. Also ist im obigen Beispiel Q ein Körper. Wir testen dies mit der Funktion `IsField(Ring)`.

```
IsField(Q);
J := ideal<R | 12>;
P, p := quo<R | J>;
IsField(P);
```

Im folgenden wollen wir nun Polynomringe betrachten. Wir wollen das folgende Ziel verfolgen. Ausgehend vom Körper $F := \mathbb{F}_p$, den wir wie oben gezeigt konstruieren können, wollen wir den Körper mit $q = p^n$ Elementen konstruieren. Tatsächlich gibt es bis auf Isomporphie genau einen Körper mit q Elementen, so daß die Verwendung des Wörtchens "den" gerechtfertigt ist.

Sei nun $f(x) \in F[x]$ ein normiertes, irreduzibles Polynom vom Grad n. Dann wissen wir aus der Vorlesung, daß $F[x]/(p(x))$ ein Körper mit q Elementen ist. Wiederholen Sie bitte den Beweis dazu.

Wenn wir also einen Körper mit q Elementen konstruieren wollen, so müssen wir ein normiertes, irreduzibles Polynom vom Grad n finden. Dazu wählen wir ein zufälliges normiertes Polynom vom Grad n und testen es auf Irreduzibilität. Dazu definieren wir zunächst den Polynomring und schreiben dann eine Intrinsic, die ein solches zufälliges Polynom generiert.

```
p := 19;
F := GF(p);
FX<x> := PolynomialRing(F);
```

Der Befehl `GF` steht für GaloisField und erzeugt den Körper $\mathbb{F}_p$. Durch `FX := PolynomialRing(F);` erzeugt man den Polynomring über F und kann nun die Unbekannte mit `FX.1` ansprechen. Durch `FX<x> := PolynomialRing(F);` wird automatisch $x := FX.1$ gesetzt und wir können Polynome in der uns vertrauten Weise schreiben.

Hier ist nun die Intrinsic, die ein zufälliges Polynom vom Grad n generiert.

```
intrinsic RandomPoly(FpX :: RngUPol, n :: RngIntElt) -> RngUPolElt
{Zufaelliges normiertes Polynom vom Grad n ueber dem endlichen Koerper F.}
    x := FpX.1;
```

```
    Fp := CoefficientRing(FpX);
    q := x^n;
    for i:=0 to n-1 do
        q := q + Random(Fp)*x^i;
    end for;
    return q;
end intrinsic;
```

Der Typ `RngUPol` steht für Ring der univariaten Polynome, d.h. Polynome in einer Variablen. Wir wollen ein Polynom zurückgeben, also ein Objekt vom Typ `RngUPolElt`. Die Funktion ist wieder weitgehend selbsterklärend. Wenn Sie genaueres über die verwendeten Intrinsics wissen wollen, schauen Sie bitte im Handbuch nach.

Nachdem wir nun zufällige Polynome erzeugen können, schreiben wir eine Funktion, die ein irreduzibles, normiertes Polynom vom Grad n findet. Das Verfahren ist rein heuristisch und wir wollen uns im Anschluss klar machen, warum das relativ gut funktioniert.

```
intrinsic IrrPoly(FpX :: RngUPol, n) -> RngUPolElt
{Sucht ein irreduzibles Polynom vom Grad n.}
    q := RandomPoly(FpX, n);
    while not IsIrreducible(q) do
        q := RandomPoly(FpX, n);
    end while;
    return q;
end intrinsic;
```

Wir generieren also so lange Polynome, bis wir ein irreduzibles gefunden haben. Zum Test auf Irreduzibilität verwenden wir dazu die Intrinsic `IsIrreducible`.

Aufgabe 6.1. Ändern Sie die Funktion so ab, dass auch die Anzahl der Versuche zurückgegeben wird. Ermitteln Sie empirisch die mittlere Anzahl der Versuche.

Aufgabe 6.2. Die Wahrscheinlichkeit, dass ein zufällig gewähltes normiertes Polynom vom Grad n über $\mathbb{F}_p$ irreduzibel ist, geht für $p \longrightarrow \infty$ gegen $\frac{1}{n}$. Beweisen Sie dies für $n = 2$ und $n = 3$.

Nachdem wir nun ein irreduzibles Polynom gefunden haben, können wir auch einen endlichen Körper mit $q = p^n$ Elementen konstruieren. Hier ein Beispiel.

```
p := 37;
n:=3;
F := GF(p);
FX<x> := PolynomialRing(F);
f := IrrPoly(FX, n);
K := quo<FX | f>;
IsField(K);
#K eq p^n;
```

Aus der Vorlesung ist bekannt, daß die multiplikative Gruppe eines endlichen Körpers zyklisch ist. Elemente $w \in K^\times$, die $K^\times$ erzeugen, nennt man Primitivwurzeln. Wir wollen im folgenden eine solche Primitivwurzel bestimmen. Es gilt:

$$w \text{ ist Primitivwurzel } \iff \operatorname{ord}(w) = q - 1, \text{ wobei } q = |K|.$$

Da uns nichts besseres einfällt, gehen wir wieder heuristisch vor. Mit `w := Random(K);` erhalten wir ein zufälliges Element in K. Falls $w \neq 0$ gilt, so wollen wir testen, ob $\mathrm{ord}(w) = q-1$ gilt. Dazu faktorisieren wir $q-1$,

$$q - 1 = p_1^{e_1} \cdots p_r^{e_r},$$

mit paarweise verschiedenen Primzahlen $p_1, \ldots, p_r$ und Exponenten $e_1, \ldots, e_r \in \mathbb{N}$. Dann gilt:

$$\mathrm{ord}(w) = q - 1 \iff w^{(q-1)/p_i} \neq 1 \text{ für } i = 1, \ldots, r.$$

Aufgabe 6.3. Beweisen Sie diesen Sachverhalt und benutzen Sie ihn, um eine Funktion zu schreiben, die eine Primitivwurzel findet.

Aufgabe 6.4. Sei r die Anzahl der Primteiler von $q-1$. Zeigen Sie, daß die Wahrscheinlichkeit, daß ein zufälliges $w \in K^\times$ eine Primitivwurzel ist, größer als $\frac{1}{2^r}$ ist. Hinweis: Es gibt $\varphi(q-1)$ verschiedene Erzeuger von $K^\times$, wobei hier φ die Eulersche phi-Funktion bezeichnet.

7. RSA

Wir wollen uns zunächst das RSA-Verfahren in Erinnerung rufen.

Seien p, q zwei verschiedene Primzahlen und $N = pq$. Der Klartext sei kodiert in $\mathbb{Z}/N\mathbb{Z}$. Sei $M := \varphi(N) = (p-1)(q-1)$ die Ordnung von $(\mathbb{Z}/N\mathbb{Z})^\times$, wobei hier gemäß Definition φ die Eulersche phi-Funktion bezeichnet. Wir wählen nun $e \in \mathbb{N}$ mit $\mathrm{ggT}(M, e) = 1$ und veröffentlichen N und e. Der Klartext wird mittels $c \mapsto y := c^e$ verschlüsselt.

Zum Entschlüsseln bestimmen wir $d \in \mathbb{N}$, so dass $ed \equiv 1 \pmod{M}$ gilt. Wenn M bekannt ist, so kann man dies mit dem Euklidischen Algorithmus machen: man bestimme $a, d \in \mathbb{Z}$ mit $1 = aM + de$, dann erfüllt $d \bmod M$ die geforderte Bedingung $ed \equiv 1 \pmod{M}$. Nach dem Satz von Lagrange folgt dann für $y \in (\mathbb{Z}/N\mathbb{Z})^\times$

$$y^d \equiv c^{ed} = c^{1+qM} = cc^{qM} \equiv c \pmod{N}.$$

Aufgabe 7.1. Zeigen Sie, dass $y^d \equiv c \pmod{N}$ für alle y gilt.

Die Sicherheit des RSA-Verfahrens beruht also darauf, dass ein Angreifer aus der Kenntnis von e und N nicht in der Lage ist, $M = (p-1)(q-1)$ zu bestimmen. Letztlich beruht also die Sicherheit darauf, dass es keinen Algorithmus gibt, der es ermöglicht, sehr große Zahlen N zu faktorisieren.

Zunächst wollen wir eine intrinsic schreiben, die einen Klartext umwandelt in eine Sequenz von Zahlen in $\mathbb{Z}/N\mathbb{Z}$. Sei dazu $B := 256$ und $l \in \mathbb{N}$, so dass $B^l < N$. Ein alphanumerische Klartext wird nun zu Gruppen von je l Zeichen zusammengefasst. Jeder solchen Zeichenkette $x := x_1 x_2 \ldots x_l$ der Länge l wird die Zahl $W(x) := w(x_1) \cdot B^{l-1} + w(x_2) \cdot B^{l-2} + \ldots w(x_{l-1}) \cdot B + w(x_l) \pmod{N}$ zugeordnet, wobei

$$w: \text{ Menge der zulässigen Zeichen} \longrightarrow \{1, \ldots, B-1\}$$

jedem Zeichen seinen ASCII-Code zuordnet.

Betrachten wir dazu folgende Befehle

```
s := "abcde";
t := "fgh";
s cat t;
Reverse(s);
#s;
```

```
StringToCode(s[1]);
for i:=1 to #s do print s[i], "---->", StringToCode(s[i]); end for;
```

Die intrinsic `StringToCode` berechnet also zu einem Zeichen seinen ASCII-Code. Umgekehrt kann man mittels `CodeToString` vom ASCII-Code wieder auf das Zeichen zurückrechnen.

Die folgenden intrinsics kodieren nun einen Zeichenkette in eine Sequenz von Zahlen in $\mathbb{Z}/N\mathbb{Z}$ wie oben beschrieben.

```
intrinsic MessageToCode(s :: MonStgElt, l :: RngIntElt, B :: RngIntElt) -> SeqEnum
{}
   if (#s mod l) ne 0 then
       t :="";
       for i:= (#s mod l) to l-1 do
           t := t cat "*";
       end for;
       s := s cat t;
   end if;

   c := [];
   for i:=0 to Integers()!(#s/l - 1) do
       x := 0;
       for j:=1 to l do
           x := x + StringToCode(s[i*l+j]) * B^(l-j);
       end for;
       Append(~c, x);
   end for;
   return c;
end intrinsic;
```

Versuchen Sie folgendes Beispiel

```
s := "abc";
c := MessageToCode(s,3,128);
```

Das Ergebnis ist $c = 97B^2 + 98B + 99$.

Die folgende intrinsic macht aus einer Sequenz von Zahlen in $\mathbb{Z}/N\mathbb{Z}$ wieder einen String.

```
intrinsic CodeToMessage(c :: SeqEnum, l :: RngIntElt, B :: RngIntElt) -> MonStgElt
{}
    s := "";
    for w in c do
        t := "";
        y := w;
        for i:=0 to l-1 do
            x := y mod B;
            if x eq 0 then
                 print "Kein Codewort !";
                 return "ERROR";
            end if;
            t := t cat CodeToString(x);
            y := Integers() ! ( (y-x)/B );
```

```
        end for;
        s := s cat Reverse(t);
    end for;
    return s;
end intrinsic;
```

Testen Sie die intrinsics an verschiedenen Beispielen, etwa:

```
B := 128;
l := 5;
s := "Ich bin der Hans Moser und lerne Computeralgebra";
c := MessageToCode(s, l, B);
c;
t := CodeToMessage(c, l, B);
t;
```

Die folgende intrinsic verschlüsselt nun gemäß dem RSA-Verfahren.

```
intrinsic RSA(c :: SeqEnum, e :: RngIntElt, N :: RngIntElt) -> SeqEnum
{}
    R, f := quo< Integers() | N >;
    return [Integers() ! (f(w)^e) : w in c];
end intrinsic;
```

Versuchen Sie zum Beispiel:

```
p := NextPrime(3246983214);
q := NextPrime(21347687621);
N := p*q;
e := NextPrime(217689);

v := RSA(c,e,N);
```

Aufgabe 7.2. Es sei

```
N := 10000003573637866544540128419343;
e := 1237;
l := 7;
B := 256;
```

Entschlüsseln Sie die folgende Nachricht:

```
v := [ 32288995028606508813339840354 34, 22019489775416062353207022839 44,
64210319346190194892047329110 7, 98205194111362512874995660268 24,
28408957707572906034791312896 72, 83923220804968755276283653880 83 ];
```

Schreiben Sie dazu eine Entschlüsselungsroutine.

8. Zahlkörper

Falls k ein Körper ist und $f \in k[x]$ ein irreduzibles Polynom, so ist bekannterweise $k[x]/(f)$ ein Erweiterungskörper von k. Etwas genauer: oE können wir annehmen, dass f normiert ist,

$$f(x) = x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 \text{ mit } a_i \in k.$$

Sei $\omega \in E$ eine Nullstelle von f, wobei E ein geeigneter Erweiterungskörper von k ist, z. B. $E = \mathbb{C}$ falls $k = \mathbb{Q}$, oder im Allgemeinen kann man stets einen algebraischen

Abschluss von k nehmen. Dann ist

$$\varphi\colon k[x]/(f) \longrightarrow k(\omega), \quad g(x) + (f) \mapsto g(\omega), \tag{1}$$

ein Isomorphismus von Körpern mit $\tau(a) = a$ für alle $a \in k$. Diesen Sachverhalt haben wir schon an früherer Stelle benutzt, etwa bei der Konstruktion von endlichen Körpern mit $q = p^n$ Elementen, wobei p eine Primzahl ist.

In diesem Abschnitt ist k stets gleich dem Körper der rationalen Zahlen $\mathbb{Q}$ oder eine endliche Erweiterung von $\mathbb{Q}$. Solche Erweiterungen nennt man Zahlkörper oder englisch *number field*. Wegen ihrer Bedeutung in der Zahlentheorie bietet MAGMA eine ganze Reihe von Funktionen, die den Umgang mit Zahlkörpern ermöglichen.

Wir beginnen mit den einfachsten Zahlkörpern, den quadratischen Erweiterungen von $\mathbb{Q}$. Die folgende Sequenz erzeugt den Körper $\mathbb{Q}(\sqrt{d})$, wobei wie üblich $d \in \mathbb{Z} \setminus \{0, 1\}$ quadratfrei ist.

```
QX<x> := PolynomialRing(Rationals());
d := 2;
f := x^2 - d;
K<om> := NumberField(f);
```

Hierdurch wurde der Körper $K := \mathbb{Q}[x]/(x^2-d)$ erzeugt, wobei MAGMA implizit stets die Isomorphie (1) benutzt. Die Elemente in K sind also von der Form $a + b\omega$ mit $a, b \in \mathbb{Q}$. Wir können nun in K die üblichen Rechenoperationen durchführen.

```
(179182 + 4321*om) / (4321961236418724 + 21387619832*om);
om^50;
Eltseq(1+2*om);
Eltseq(1+3*om);
MinimalPolynomial(2 + om);
MinimalPolynomial(om);
```

Der Befehl `MinimalPolynomial` liefert, wie zu erwarten, das Minimalpolynom. Verifizieren Sie die Ergebnisse.

Aufgabe 8.1. Schreiben Sie eine intrinsic

```
intrinsic MyQuadMipo(alpha :: FldNumElt) -> RngUPolElt
```

die das Minimalpolynom zu $\alpha \in K$ berechnet. Vergleichen Sie die Ergebnisse Ihrer Funktion mit denen von `MinimalPolynomial`.

Wir wollen nun die Gruppe der $\mathbb{Q}$-Automorphismen $\mathrm{Aut}(K/\mathbb{Q})$ von $K/\mathbb{Q}$ bestimmen. Hierbei ist

$$\mathrm{Aut}(K/\mathbb{Q}) = \{\tau\colon K \longrightarrow K \mid \tau \text{ ist } \mathbb{Q} - \text{Homomorphismus mit } \tau|_{\mathbb{Q}} = id\}.$$

Im Fall quadratischer Zahlkörper kennen wir diese Gruppe bereits aus der Vorlesung. Es gilt:

$$\mathrm{Aut}(K/\mathbb{Q}) = \{id, \tau\}, \text{ wobei } \tau(a + b\omega) = a - b\omega \text{ gilt.}$$

Man beachte, dass τ bereits durch $\tau(\omega) = -\omega$ vollständig bestimmt ist.

MAGMA stellt uns hierfür, auch für beliebige Zahlkörper, die Routine `Automorphisms` zur Verfügung. Führen Sie die folgenden Experimente durch:

```
Aut := Automorphisms(K);
Aut;
tau1 := Aut[1];
tau2 := Aut[2];
```

```
Type(tau1);
Domain(tau1);
Codomain(tau1);

tau1(om);
tau2(om);
tau2(12222 + 12*om);
tau1(12222 + 12*om);
```

Im Weiteren wollen wir nun den Körper $L := \mathbb{Q}(\sqrt{d_1}, \sqrt{d_2})$ mit $d_1, d_2 \in \mathbb{Z} \setminus \{0, 1\}$ quadratfrei und $d_1 \neq d_2$ konstruieren. Betrachten Sie dazu die folgende Befehlssequenz.

```
d1 := 3;
d2 := 5;
QX<x> := PolynomialRing(Rationals());
K<om1> := NumberField(x^2 - d1);
KX<y> := PolynomialRing(K);
L<om2> := NumberField(y^2 - d2);
```

Es sei $G := \mathrm{Aut}(L/\mathbb{Q})$. Es gilt:

$$G = \{id, \tau, \sigma, \sigma\tau\} \tag{2}$$

mit

$$\sigma(\omega_1) = -\omega_1, \quad \sigma(\omega_2) = \omega_2,$$
$$\tau(\omega_1) = \omega_1, \quad \tau(\omega_2) = -\omega_2,$$

Wir wollen nun zu einem $\alpha \in L$ das Polynom

$$h(x) := \prod_{\tau \in G} (x - \tau(\alpha))$$

berechnen.

Aufgabe 8.2. Sei $\alpha \in L$.

a) Zeigen Sie: α ist von der Form

$$\alpha = c_1 + c_2\omega_1 + c_3\omega_2 + c_4\omega_1\omega_2.$$

mit eindeutig bestimmten $c_1, \ldots, c_4 \in \mathbb{Q}$.

b) Zeigen Sie: $\alpha \in \mathbb{Q} \iff \sigma(\alpha) = \alpha$ und $\tau(\alpha) = \alpha$.

c) Zeigen Sie: $h \in \mathbb{Q}[x]$.

Ein beliebiges Element in L ist von der Form $r + s\omega_2$ mit $r, s \in K$. Also sind r, s von der Form $r = a_1 + b_1\omega_1$ bzw. $s = a_2 + b_2\omega_1$. Wir führen die folgenden Experimente durch:

```
om2^2;
om1^2;
alpha := om1 + om2;
coeff := Eltseq(alpha); coeff;
r := coeff[1];
s := coeff[2];
coeff := Eltseq(r); coeff;
a1 := coeff[1];
```

```
b1 := coeff[2];
coeff := Eltseq(s); coeff;
a2 := coeff[1];
b2 := coeff[2];
LX<z> := PolynomialRing(L);
alpha1 := (a1-b1*om1) + (a2-b2*om1)*om2;
alpha2 := (a1+b1*om1) - (a2+b2*om1)*om2;
alpha3 := (a1-b1*om1) - (a2-b2*om1)*om2;
LX<z> := PolynomialRing(L);
h := (z - alpha) * (z - alpha1) * (z - alpha2) * (z - alpha3);
```

Aufgabe 8.3. Entwickeln Sie nun eine Funktion

```
intrinsic MyBiquadMipo(alpha :: FldNumElt, om1 :: FldNumElt, om2 :: FldNumElt) -> RngUPolElt
```

die das Minimalpolynom zu $\alpha \in L$ berechnet.

Aufgabe 8.4. Zeigen Sie, dass $\omega_1 + \omega_2$ ein primitives Element von $L/\mathbb{Q}$ ist.

Das Minimalpolynom von $\beta := \omega_1 + \omega_2$ ist durch $f(x) = x^4 - 16x^2 + 4$ gegeben. Das sollte sich bei der Bearbeitung der letzten Aufgabe ergeben haben. Wir wollen nun den Körper $L = \mathbb{Q}(\sqrt{d_1}, \sqrt{d_2})$ in einem Schritt erzeugen. Dazu gehen wir wie folgt vor.

```
f := x^4 - 16*x^2 + 4;
F<beta> := NumberField(f);
```

Wir wissen, dass ω_1 und ω_2 in F enthalten sind; allerdings ist a priori nicht klar, wie diese Elemente als $\mathbb{Q}$-Linearkombination der Basis $1, \beta, \beta^2, \beta^3$ dargestellt werden. Wir wollen nun diese Darstellungen berechnen.

```
FX<z> := PolynomialRing(F);
v := Roots(z^2 - d1); v;
w := Roots(z^2 - d2); w;
om1 := v[1,1]; om1;
om2 := w[1,1]; om2;
om1^2;
om2^2;
```

Mit der MAGMA-Funktion `Automorphisms` berechnen wir nun die Gruppe $G := \mathrm{Aut}(F/\mathbb{Q})$ und führen verschiedene Experimente durch.

```
G := Automorphisms(F);
G;
G[1](om1);
G[1](om1) eq om1;
G[1](om2) eq om2;
G[2](om2) eq om2;
G[2](om2) eq -om2;
G[2](om1) eq -om1;
```

Aufgabe 8.5. Ordnen Sie `G[1], G[2], G[3], G[4]` den Elementen $id, \sigma, \tau, \sigma\tau$ aus (2) zu.

9. Zerfällungskörper

In diesem Abschnitt wollen wir den Zerfällungskörper von $K = \mathbb{Q}(\sqrt[p]{m})$ studieren, wobei p eine Primzahl und $m \in \mathbb{N} \setminus \{1\}$ quadratfrei ist. Wir setzen

$$\omega := \sqrt[p]{m} \in \mathbb{R}.$$

Dann ist das Minimalpolynom von ω gegeben durch $f(x) = x^p - m$, was man sofort durch Anwendung des Eisensteinkriteriums einsieht. Wir realisieren dies in MAGMA.

```
QX<x> := PolynomialRing(Rationals());
p := 5;
m := 13;
f := x^p - m;
K<om> := NumberField(f);
MinimalPolynomial(om);
```

Es sei $\zeta = \exp(2\pi i/p)$. Dann sind die Nullstellen von f gegeben durch $\zeta^i\omega$ mit $i = 0, \ldots, p-1$.

Aufgabe 9.1. Sei L der Zerfällungskörper von f.

a) Zeigen Sie: $L = \mathbb{Q}(\omega, \zeta)$.

b) Zeigen Sie: Das Minimalpolynom von ζ über K ist gegeben durch

$$g(x) = x^{p-1} + x^{p-2} + \ldots + x + 1.$$

c) Realisieren Sie L in MAGMA.

Die Gruppe $G := \mathrm{Aut}(L/K)$ ist gegeben durch

$$G = \{\sigma_{ij} \mid 1 \leq i \leq p-1, 0 \leq j \leq p-1\},$$

wobei die Automorphismen σ_{ij} definiert sind durch

$$\sigma_{ij}(\omega) = \zeta^j\omega, \quad \sigma_{ij}(\zeta) = \zeta^i. \tag{3}$$

Im Weiteren wollen wir nun das Minimalpolynom von $\alpha := \omega + \zeta$ bestimmen. Dazu beobachten wir zunächst, dass

$$\sigma_{ij}(\zeta + \omega) = \zeta^i + \zeta^j\omega$$

ist. Betrachten Sie die folgende Aufgabe.

Aufgabe 9.2. Gehen Sie die folgende Befehlssequenz Zeile für Zeile durch und erläutern Sie die Befehle und deren Ergebnisse.

```
QX<x> := PolynomialRing(Rationals());
p := 5; m := 3;
f := x^p - m;
IsIrreducible(f);
K<omega> := NumberField(f);
Automorphisms(K);
KX<y>:= PolynomialRing(K);
Factorization(f);
Factorization(KX!f);
g := &+[y^i : i in [0..p-1]]; g;
L<zeta> := NumberField(g);
LX<z>:= PolynomialRing(L);
```

```
fac := Factorization(LX!f);
fac;
zeta^p;
omega^p;
conj := [zeta^i + zeta^j*omega : i in [1..p-1], j in [0..p-1]];
h := &*[z-c : c in conj]; h;
h := QX ! h;

IsIrreducible(h);
```

Wie im Beispiel der biquadratischen Zahlkörper wollen wir nun den Körper $L = \mathbb{Q}(\alpha) = \mathbb{Q}(\omega, \zeta)$ in einem Schritt erzeugen. Sodann identifizieren wir ω und ζ als Elemente $\mathbb{Q}(\alpha)$, dargestellt in der Basis $1, \alpha, \ldots, \alpha^{n-1}$ mit $n = p(p-1)$.

```
F<alpha> := NumberField(h);

FX<t> := PolynomialRing(F);
f := t^p - m;
nullst := Roots(f); nullst;
omega := nullst[1,1];
omega^p;
omega;
g := &+[t^i : i in [0..p-1]];
nullst := Roots(g); nullst;
zeta := nullst[1,1];
zeta^p;
zeta;
```

Aufgabe 9.3. Durch den Befehl `G := Automorphisms(F);` erhalten wir die Gruppe $\mathrm{Aut}(F/\mathbb{Q})$. Ordnen Sie wie in Aufgabe 8.5 den Automorphismen σ_{ij} aus (3) die Elemente `G[k]` zu. Schreiben Sie dazu eine intrinsic

```
intrinsic WhichAut(G :: SeqEnum, i :: RngIntElt,
                   j :: RngIntElt, p :: RngIntElt, m :: RngIntElt) -> Map
```

10. Galoistheorie

Durch die folgende Befehlssequenz erzeugen wir den Zahlkörper zum irreduziblen Polynom

$$h(x) = x^9 - 85x^8 + 2188x^7 - 16561x^6 + 48471x^5 - 61901x^4 + 32467x^3 - 4542x^2 + 132x - 1.$$

```
QX<x> := PolynomialRing(Rationals());
h := x^9 - 85*x^8 + 2188*x^7 - 16561*x^6 + 48471*x^5
     - 61901*x^4 + 32467*x^3 - 4542*x^2 + 132*x - 1;
E := NumberField(h);
```

Wir wollen zunächst experimentell feststellen, ob $E/\mathbb{Q}$ galoissch ist. Dazu reicht es zu zeigen, dass $E/\mathbb{Q}$ normal ist. Dafür wiederum reicht es zu zeigen, dass $G(E/\mathbb{Q}) = [E : \mathbb{Q}] = \deg(h)$ gilt. Wie wir bereits wissen, berechnet man $G(E/K)$ mit dem Befehl `Automorphisms`.

```
G := Automorphisms(E);
#G eq Degree(h);
```

Für galoissche Erweiterungen $E/\mathbb{Q}$ stellt MAGMA die intrinsic `AutomorphismGroup` bereit. Testen Sie:

```
G, Aut, f := AutomorphismGroup(E);
G; Aut; f;
Type(G);
```

Die Anwendung eines Automorphismus $\sigma \in G$ auf ein Element $\alpha \in E$ ist etwas gewöhnungsbedürftig. Betrachten Sie die folgenden Befehle:

```
alpha := 1+E.1;
conjugates := [f(g)(alpha) : g in G];
&+conjugates;
&*conjugates;
```

Aufgabe 10.1. Erläutern Sie diese vier Zeilen. Gehen Sie insbesondere darauf ein, warum die Summe der Konjugierten sowie das Produkt der Konjugierten in $\mathbb{Q}$ liegen.

Wir wollen im Weiteren die sämtlichen Zwischenkörper von $E/\mathbb{Q}$ bestimmen, indem wir für jeden dieser Zwischenkörper ein primitives Element angeben. Dazu bestimmen wir die sämtlichen Untegrgruppen U von G. Die Zwischenkörper sind dann durch die Fixkörper E^U gegeben. G hat die Ordnung 9, ist also abelsch (man erinnere sich: Gruppen der Ordnung p^2 sind stets abelsch). Es stellt sich also nur die Frage, ob $G \simeq C_3 \times C_3$ oder $G \simeq C_9$ ist. Wir testen dies wie folgt.

```
IsAbelian(G);
IsCyclic(G);
```

G ist also vom Typ $C_3 \times C_3$. Man zeigt leicht, dass Gruppen vom Typ $C_p \times C_p$ genau $p+1$ verschiedene Untergruppen der Ordnung p haben (Beweis!). Die Untergruppen von G (bis auf Konjugation) erhält man durch den Befehl `Subgroups(G)`. Dieser Befehl liefert eine Liste von Records zurück. Die folgende Sequenz von Befehlen zeigt, wie man die jeweiligen Untergruppen erhält.

```
S := Subgroups(G);
S;
U := S[2]'subgroup;
```

Aufgabe 10.2. Berechnen Sie primitive Elemente β_i für die vier Zwischenkörper F_i von $E/\mathbb{Q}$ mit $[F_i : \mathbb{Q}] = 3$. Exprimentieren Sie dazu mit Elementen der Form

$$\sum_{g \in U} g(\alpha) \text{ oder } \prod_{g \in U} g(\alpha)$$

für verschiedene $\alpha \in E$.

Abschließend wollen wir endliche Körper betrachten. Die folgende Befehlssequenz erzeugt die Körpererweiterung $\mathbb{F}_q/\mathbb{F}_p$ mit $q = p^d$.

```
p := 5;
d := 10;
q := p^d;
Fp := GF(p);
K := GF(q);
```

Aus der Algebra ist bekannt, dass $\mathbb{F}_q/\mathbb{F}_p$ galoissch ist mit $G(\mathbb{F}_q/\mathbb{F}_p) = \langle \sigma_p \rangle$, wobei σ_p den Frobeniusautomorphismus bezeichnet, d.h. $\sigma_p(\alpha) = \alpha^p$ für alle $\alpha \in \mathbb{F}_q$. Wir

wollen nun wie oben für die sämtlichen Zwischenkörper ein primitives Element und das entsprechende Minimalpolynom dazu angeben. Da die Galoisgruppe zyklisch von der Ordnung d ist, gibt es zu jedem Teiler f von d genau eine Untegruppe U mit $|U| = d/f$, nämlich $U = \langle \sigma_p^f \rangle$. Also gibt es gemäß dem Hauptsatz der Galoistheorie zu jedem Teiler f von d genau einen Zwischenkörper F mit $[F : \mathbb{F}_p] = f$, nämlich $F = \mathbb{F}_q^U$.

Betrachten und verstehen Sie die folgende Sequenz von Befehlen.

```
p := 5;
d := 10;
q := p^d;
Fp := GF(p);
FpX<x> := PolynomialRing(Fp);
K := GF(q);
Kmal, j := UnitGroup(K);
alpha := j(Kmal.1);
MinimalPolynomial(alpha);

f := 5;
m := (Integers() ! (d/f));

beta1 := &+[alpha^(p^(f*i)) : i in [0..(m-1)]];
MinimalPolynomial(beta1);

n := Integers()!((p^d-1)/(p^f-1));
beta2 := alpha^n;
MinimalPolynomial(beta2);
```

Bei der Berechnung von β_1 gehen wir prinzipiell wie im ersten Beispiel vor und bilden eine geeignete Summe über verschiedene Konjugierte. Genauer:

$$\beta_1 = \sum_{\sigma \in \langle \sigma_p^f \rangle} \sigma(\alpha).$$

Falls wir wie oben $U := \langle \sigma_p^f \rangle$ setzen, so ist $\beta_1 \in K^U$ und $K^U/\mathbb{F}_p$ ist galoissch mit Gruppe

$$G(K^U/\mathbb{F}_p) \simeq G(K/\mathbb{F}_p)/U.$$

Also ist $[K^U : \mathbb{F}_p] = |G(K/\mathbb{F}_p)|/|U| = \frac{d}{d/f} = f$. Falls also das Minimalpolynom von β_1 den Grad f hat, so ist β_1 ein primitives Element.

Bei der Berechnung von β_2 nutzen wir noch mehr die spezielle Struktur endlicher Körper aus. Das Element α ist ein Erzeugendes von $K^\times$, hat also die genaue Ordnung $p^d - 1$. Also hat das Element $\beta_2 := \alpha^n$ mit $n := \frac{p^d-1}{p^f-1}$ die genaue Ordnung $p^f - 1$, erzeugt also den endlichen Körper mit p^f Elementen.

Aufgabe 10.3. Schreiben Sie eine intrinsic, die zu einem endlichen Körper K mit $\mathrm{char}(K) = p > 0$ und einem Teiler f von $d := [K : \mathbb{F}_p]$ ein primitives Element β für den Teilkörper L von $K/\mathbb{F}_p$ mit $|L| = p^f$ und das zugehörige Minimalpolynom berechnet.

LMU München, Theresienstrasse 39, 80333 München, Germany
Email address: bley@math.lmu.de