Computing with Magma

David A. Craven

June 9, 2008

Contents

Chapter 1

The First Steps

To start Magma, type `magma` into a terminal window. After doing this, one of two things will happen: either the screen will display something like

```
Magma V2.13-15    Wed Jan  2 2008 20:09:11 on crazylegs-crane  [Seed = 255043391]
Type ? for help.  Type <Ctrl>-D to quit.
>
```

and you may begin, or you will see something like

```
Magma is not authorised for use on this machine.

This host has the following MAC address(es):
00:1d:60:9a:7f:dd
Please contact Magma at magma@maths.usyd.edu.au on the Internet.
```

and you need to log in to a different computer. [This is due to the licensing restrictions imposed on Magma by its creators.]

The Magma on-line help can be found at

`http://magma.maths.usyd.edu.au/magma/htmlhelp/Magma.htm`.

1.1 Arithmetic and Data Types

Now we are all at the same place, let us try a few commands. We begin with simple arithmetic:

```
> 2+4;
6
> 3*7;
21
> 5/7;
5/7
```

```
> 3^2;
9
> 31 mod 5;
1
```

[Note that each individual command is ended with a semicolon; this is essential for Magma to understand where the command ends, as Magma does not assume that a pressed 'Enter' key signifies the end of a command.]

Magma believes that values like 2, 4, 0, and so on, are *integers*, and will treat them as such, until you give it a command that makes it treat them as something else, like `5/7`, which makes Magma turn them into another thing, *rationals*. Suppose that you actually want to know the decimal expansion of 5/7: then you need to turn it into a *real number*, which requires a decimal somewhere, such as

```
> 5/7.0;
0.714285714285714285714285714285
```

We can assign numbers to letters, and do arithmetic with the letters like we did with the numbers.

```
> x:=2;
> x^2;
4
> x:=x+1;
> x;
3
> x+:=1;
> x;
4
> 3*x;
12
```

(Note here that to return $3x$, you must input `3*x`.) The last but one command is a (slightly) quicker way of altering a variable already in memory. There are similar commands for the other arithmetic operations.

```
> x:=10;
> x-:=2; x;
8
> x/:=4; x;
2
> y:=7;
> x*:=y; x;
14
```

[Here more than one command has been entered on one line, to save space: this is perfectly acceptable.]

Now we come to our first *function*, the `Type` function. This tells you what type of object you have. Earlier, we said that Magma stored `4` as an integer, `5/7` as a rational, and `5.0` as a real number. We can test this claim using the `Type` function.

```
> x:=4; Type(x);
RngIntElt
> x:=5/7; Type(x);
FldRatElt
> x:=5.0; Type(x);
FldReElt
```

This can cause problems, as the next example shows.

```
> x:=10/3; y:=3/2;
> x*y;
5
> 5 mod 4;
1
> x*y mod 4;

>> x*y mod 4;


Runtime error in 'mod': Bad argument types
Argument types given: FldRatElt, FldRatElt
```

What went wrong? The answer is that `mod` can only work with integers, but Magma thinks that since `x` and `y` are rationals—objects with type `FldRatElt`—the product `x*y` is also a rational, even though it is displayed as an integer. This is why it's important to understand what the types of objects are. Getting around this problem of Magma storing integers as rationals will be tackled in the next section.

This can also occur for simple integer division.

```
> x:=4; y:=2;
> Type(x/y);
FldRatElt
```

Suppose that we know that `y` divides `x`; in this case, we can do integer division using the command `div`.

```
> 4 div 2;
2
> 5 div 2;
2
```

This finishes off the obvious arithmetic operations that are used. Later on, we will look at primes, factorizations, and rings that are not $\mathbb{Z}$, but for now this will do. Ending this section, we ought to learn how to end Magma; this can be done with either `quit;` or `exit;`.

1.2 Sets, Subsets, and Belonging

In the previous section, we learnt about data types. We saw three data types: `RngIntElt`; `FldRatElt`; and `FldReElt`. There is one more type of number conspicuously absent from that list.

```
> Sqrt(-1);
1.00000000000000000000000000000*$.1
> Type(Sqrt(-1));
FldComElt
```

Thus we now have four types of objects. There are many more types of objects, and we will encounter some of them during the course. Types have another name in Magma, that of *categories*. It is the type of data types, so to speak.

```
> Type(RngIntElt);
Cat
```

So the type of types is `Cat`.

We have constructed all types of numbers that we care about, so now let's construct the rings of numbers to which they belong. The integers are constructed using the command `IntegerRing()` or `Integers()`, the rationals by `RationalField()` or `Rationals()`, the reals by `RealField()`, and the complexes by `ComplexField()`.

```
> Z:=IntegerRing(); Q:=Rationals(); R:=RealField(); C:=ComplexField();
> Z;
Integer Ring
> Type(Z);
RngInt
> Type(Q), Type(R), Type(C);
FldRat FldRe FldCom
```

[Here we see that if one separates commands with commas, the outputs are displayed on the same line.]

There should be a way of testing whether an object is an element of a set, and this can be accomplished with the command `in`. The command `A in B` tests whether `A` is an element of `B`, and returns a Boolean value—`true` or `false`—depending on the outcome.

```
> 1 in Z;
true
> (10/3 * 3/2) in Q;
true
> (10/3 * 3/2) in Z;
true
> 10/3 in Z;
false
```

The values `true` and `false` can be used in programs as conditions, i.e., *if* ⟨condition⟩ *then* ⟨command⟩. We will encounter commands like `if` and `for` later, but for now we will concentrate on sets.

We return to the problem encountered in the previous section. Above we are confidently assured that `10/3 * 3/2` is an element of `Z`, which represents the integers. However, we still cannot use `mod` on it, and we have to tell Magma to *cast* this rational number as an integer. Casting is only allowed when the object would actually make sense as an element of the set into which you are trying to cast it.

```
> Z:=Integers();
> Q:=Rationals();
> x:=10/3;
> y:=3/2;
> z:=x*y;
```

We can now cast `z` as an integer, an element of `Z`, and take it modulo 4.

```
> z in Z;
true
> Z!z mod 4;
1
> Z!x mod 4;

>> Z!x mod 4;
    ^
Runtime error in '!': Rational argument is not a whole integer
LHS: RngInt
RHS: FldRatElt
```

The last example was given to show what happens when you try to perform an incorrect casting. This can be checked by performing the command `IsCoercible`, and we will see it in action now.

```
> IsCoercible(Integers(),4/2);
true 2
> IsCoercible(Integers(),4/3);
false
```

Something interesting happened when we successfully coerced `4/2` into an integer: the command returned two values. To set a variable to be the output of a function, we have seen that one uses the `:=` *assignment operator*. For example,

```
> bool:=IsCoercible(Integers(),4/3);
> bool;
false
```

Notice that if the `:=` operator is used, then no output is produced on the screen. To assign two values, like those that were produced by `IsCoercible` when it can coerce the element into the set, we use the following syntax.

```
> bool,val:=IsCoercible(Integers(),4/2);
> val;
2
```

If Magma gets nothing back from this command for `val`, then it doesn't give an error, but deletes `val`.

```
> val:=2;
> bool,val:=IsCoercible(Integers(),4/3);
> val;


>> val;
   ^
User error: Identifier 'val' has not been declared or assigned
```

We end this piece on assignment by noting what happens if you try to assign more variables than there should be:

```
> a,b,c:=IsCoercible(Integers(),4/2);


>> a,b,c:=IsCoercible(Integers(),4/2);
        ^
Runtime error in :=: Expected to assign 3 value(s) but only computed 2 value(s)
```

Now we will move on to sets. A set is constructed in Magma exactly as it is written in mathematics, using braces.

```
> X:={1,2,3};
> X;
{ 1, 2, 3 }
```

Sets in Magma perform exactly as they do in mathematics, as in the set $\{1, 2, 3\}$ is the same as the set $\{1, 2, 3, 3\}$. The command `eq` tests whether two objects are equal.

```
> X:={1,2,3}; Y:={1,2,3,3}; Z:={2,3,4};
> X eq Y;
true
> X eq Z;
false
> X join Z;
{ 1, 2, 3, 4 }
> X meet Z;
{ 2, 3 }
> X diff Z;
{ 1 }
> X sdiff Z;
{ 1, 4 }
> 1 in X;
true
```

Notice that, in Magma, if the sets contain integers, then Magma naturally orders them with the usual ordering.

To find out how many elements a set contains, one uses the `#` symbol. Finally, there is a shorthand for producing the set of numbers in an arithmetic progression.

```
> A:={1..10};
> #A;
10
> 6 in A;
true
> A;
{ 1 .. 10 }
> B:={1,2,3,4,5,6,7,8,9,10};
> A eq B;
true
```

```
> C:={1..9 by 2};
> 3 in C;
true
> 4 in C;
false
```

An extremely useful method of generating sets of numbers is exhibited in the following example.

```
> XX:={x^2: x in {1..10}};
> XX;
{ 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 }
```

The syntax is clear, and this can be used to create a variety of useful sets.

We end this section with a few other commands that are useful when dealing with sets of numbers.

```
> A:={4,2,16,-3,0};
> Maximum(A);
16
> Minimum(A);
-3
> Random(A);
0
```

These three commands (also `Min` and `Max` can be used instead of `Minimum` and `Maximum`) are concerned with selecting elements of a set. We can also include and throw out elements of a set.

```
> set:={1,4,6,10,-3};
> Include(set,17);
{ -3, 1, 4, 6, 10, 17 }
> Exclude(set,10);
{ -3, 1, 4, 6 }
> set2:=Exclude(set,10);
> set2;
{ -3, 1, 4, 6 }
```

We now end with a few exercises to attempt before moving on to the next section. At the end of the chapter, we will have a list of more involved exercises to test the skills.

Exercise 1.1 Calculate the first fifty powers of 2.

Exercise 1.2 Using Magma, calculate how many numbers of the form $x^2 + 3x + 1$ with x between 0 and 1000 are also multiples of 5.

Exercise 1.3 What is the maximal element of the image of the function $f(x) = 25x - x^2 + x^3 - 2x^4$ defined over $\mathbb{Z}$?

1.3 Other Types of Sets

Sets such as {1,2,3} can be thought of as unordered lists not allowing repetitions. There are three obvious ways to generalize this: to ordered lists; to unordered lists allowing repetitions; and to ordered lists allowing repetitions. All of these possibilities are implemented in Magma.

We will begin with unordered lists allowing repetitions, or *multisets*. These can be defined by using {* *} instead of { }.

```
> X:={* 1,1,2,3 *};
> X;
{* 1^^2, 2, 3 *}
```

These are different objects to normal sets, as can be seen from their types.

```
> A:={1,2,3};
> B:={* 4,5,6 *};
> Type(A), Type(B);
SetEnum SetMulti
> A join B;

>> A join B;
     ^
Runtime error in 'join': Bad argument types
Argument types given: SetEnum[RngIntElt], SetMulti[RngIntElt]
```

Multisets can be manipulated in the same way as normal sets, except that one can no longer use the notation {* 1..4 *}, and differences can no longer be taken.

```
> A:={*1,2,3*};
> B:={* 4,5,6 *};
> A join B;
{* 1, 2, 3, 4, 5, 6 *}
> A join A;
{* 1^^2, 2^^2, 3^^2 *}
> C:={*1,2,4,5*};
> A meet C;
{* 1, 2 *}
> B diff C;
```

```
>> B diff C;
     ^

Runtime error in 'diff': Bad argument types
Argument types given: SetMulti[RngIntElt], SetMulti[RngIntElt]

> D:={*1..4*};


>> D:={*1..4*};
         ^

User error: bad syntax
> D:={* x^2: x in {-2..2} *};
> D;
{* 0, 1^^2, 4^^2 *}
> Maximum(D);
4
> Multiplicity(D,4);
2
```

Here we introduced the command `Multiplicity`, which has the obvious meaning.

The next type of set is ordered, but cannot have multiplicity. These are slightly more complicated than multisets by virtue of their having an ordering. In particular, the union operation is now no longer commutative. Ordered sets are created using `{@ @}`.

```
> X:={@1,2,3,4,5@};
> Y:={@8,7,6,5,4@};
> X join Y;
{@ 1, 2, 3, 4, 5, 8, 7, 6 @}
> Y join X;
{@ 8, 7, 6, 5, 4, 1, 2, 3 @}
> X meet Y;
{@ 4, 5 @}
> Y meet X;
{@ 4, 5 @}
```

Here we see that `join` places all those elements of the second set not in the first one after all those in the first, in the order in which they appear in the second set. The `meet` command seems to pick out all elements in the intersection, and then give them the standard ordering. Finally for these sets, since they are ordered, we can now ask which is the nth element in the ordered set.

```
> Y:={@8,7,6,5,4@};
> Y[2];
7
```

The last type of sets are supposed to be ordered and allow multiplicities. These are so different from sets that they are technically called *sequences*. These are created using square brackets. We can no longer used commands like `meet` and `join` because these should be thought of as sequences above all other things.

```
> X:=[1..10];
> X;
[ 1 .. 10 ]
> #X;
10
> Y:=[1,2,3,4,5,6,7,8,9,10];
> X eq Y;
true
> Z:=[3,6,6,2]; Z[2];
6
```

The commands for sequences are very different from those for sets given above. You can alter elements in the sequence, append and remove elements from the end, sort and reverse sequences, and more besides. They are very versatile objects. Rather than define all of the functions here, we will illustrate their obvious effects using commands.

```
> X:=[1,6,4,3,9];
> X[3]:=5;
> X;
[ 1, 6, 5, 3, 9 ]
> Append(X,8);
[ 1, 6, 5, 3, 9, 8 ]
> X:=Append(X,8);
> X;
[ 1, 6, 5, 3, 9, 8 ]
> X:=Prune(X);
> X;
[ 1, 6, 5, 3, 9 ]
> Remove(X,3);
[ 1, 6, 3, 9 ]
> Insert(X,2,17);
[ 1, 17, 6, 5, 3, 9 ]
```

```
> X;
[ 1, 6, 5, 3, 9 ]
> Reverse(X);
[ 9, 3, 5, 6, 1 ]
> X:=Reverse(X);
> X:=Sort(X);
> X;
[ 1, 3, 5, 6, 9 ]
```

This illustrates the use of `Append`, `Prune`, `Insert`, `Remove`, `Reverse`, and `Sort`. The commands `Minimum` and `Maximum` still work, but this time return both the maximal/minimal element, along with the position that the first occurrence of that number occupies.

```
Y:=[2,5,10,10,4,10,3];
> Maximum(Y);
10 3
> a,b:=Maximum(Y);
> Y[b];
10
```

Two sequences may be concatenated, and this also offers an alternative to using the `Append` function.

```
> A:=[2,5,3];
> B:=[7,7,9,1,14];
> C:=A cat B;
> C;
[ 2, 5, 3, 7, 7, 9, 1, 14 ]
> A1:=Append(A,5);
> A2:=A cat [5];
> A1 eq A2;
true
```

Finally in this section, we show how to move from a sequence to its underlying set and back again.

```
> A;
[ 2, 5, 3, 3 ]
> SequenceToSet(A);
{ 2, 3, 5 }
> B:=SequenceToSet(A); B;
{ 2, 3, 5 }
```

```
> SetToSequence(B);
[ 2, 3, 5 ]
```

Exercise 1.4 Construct the sequence of all integers between 1 and 10. Then remove all even numbers from this sequence.

1.4 Boolean Values and Conditionals

These two sections and the first section of the next chapter contribute the main ingredients to Magma's success as a package. The first is the ability to decide whether a statement is true or false, and the second is the ability to iterate over sets. We will examine booleans in this section, and iteration in the next.

We have already seen some of Magma's potential for knowing whether two objects are equal, using the command `eq`, and another example is

```
> 4/2 eq 2;
true
```

There are plenty of other questions we can ask about numbers and objects, and they are given in the table below:

Mathematical Operation	Command
$x = y$	`x eq y`
$x \neq y$	`x ne y`
$x \geq y$	`x ge y`
$x \leq y$	`x le y`
$x > y$	`x gt y`
$x < y$	`x lt y`
$x \in X$	`x in X`

They behave in the same way as `eq` did above, in that they return true or false. However, there is a slight problem with these commands: it comes from asking whether two completely different things are equal. We give an example.

```
> X:=[1,2];
> Y:={1,2};
> X eq Y;


>> X eq Y;
     ^

Runtime error in 'eq': Bad argument types
Argument types given: SeqEnum[RngIntElt], SetEnum[RngIntElt]
```

Thus if two objects are of different data types then you cannot compare them. The exception here is numbers; earlier we compared integers and rationals, but we can go one better.

```
> 4 gt 5.0;
false
```

Well, it seems useful to be able to test whether one number is larger than the other, but how can we actually feed this into another function? What we would like is some kind of conditional command, something that performs the operation 'if some condition is true, then do this series of commands'. Fortunately, this exists. We give a simple example.

```
> a:=3; b:=4;
> if(a gt b) then a*b; end if;
> if(b gt a) then a*b; end if;
12
```

Here we see that the command structure is `if (condition) then (commands); end if;`. Now let us examine the example above. In the first if statement, since `a` is not greater than `b`, the command `a*b` is never executed, and the output is nothing. In the second if statement, the condition `b gt a` is true, and so the command `a*b` *is* executed, and the result, `12` is displayed.

It is possible to also perform one set of commands if the condition is true, and another set of commands if the condition is false. We will see this in a useful example. To display some text we need to use the `print` command. The syntax is obvious from the example below.

```
> a:=4/3;
> Z:=Integers();
> if( IsCoercible(Z,a) ) then bool,a:=IsCoercible(Z,a);
if> print("a is coercible into the integers!");
if> else print("a is not coercible into the integers!"); end if;
a is not coercible into the integers!
> a:=4/2;
> Type(a);
FldRatElt
> if( IsCoercible(Z,a) ) then bool,a:=IsCoercible(Z,a);
if> print("a is coercible into the integers!");
if> else print("a is not coercible into the integers!"); end if;
a is coercible into the integers!
> Type(a);
RngIntElt
```

In this example, we have a rational number `a`, and if `a` is really an integer in disguise, then the if statement turns `a` into an integer. If `a` is not an integer, then this fact is displayed. The syntax is therefore

```
if (condition) then (commands); else (commands); end if;
```

When you are in the middle of an if statement, Magma helpfully reminds you by printing an `if>` instead of a normal `>`.

if statements can be *nested*; that is, placed one inside another. Suppose we want to test whether a number `x` is positive, negative, or zero. We could use code like the following.

```
> if (x gt 0) then print("x is positive");
if> else if(x lt 0) then print("x is negative");
if|if> else print("x is zero");
if|if> end if; end if;
```

Here, since if statements return true or false, we have used two if statements to deal with three mutually exclusive possibilities. In the case where this happens, instead of writing `else if`, one may use a similar command, namely `elif`. This is not a nested if statement, but merely allows more flexibility to the else routine.

The idea is the following: suppose we have several true-false tests to run, but they can be run one at a time, such as checking whether a number is positive, negative or zero. If the first one results in a positive answer, then we stop, and otherwise we keep going with the other tests until we get a positive answer or we run out of tests. We illustrate this with another example, where we want to know the congruence of `x` modulo 4.

```
> x:=5;
> if(x mod 4 eq 0) then print("x is congruent to 0 modulo 4");
if> elif(x mod 4 eq 1) then print("x is congruent to 1 modulo 4");
if> elif(x mod 4 eq 2) then print("x is congruent to 2 modulo 4");
if> else print("x is congruent to 3 modulo 4"); end if;
x is congruent to 1 modulo 4
```

[If all we wanted to do is print out the congruence of `x` modulo 4, then there are easier ways to do this. However, the print commands could be replaced by various instructions.]

Now suppose that you want to set some variable depending on whether a condition is true or false. For example, suppose that we have some non-zero variable `a`, and we want to set `s` to be the sign of `a`. This can be accomplished with the code below.

```
> if(a gt 0) then s:=1;
if> else s:=-1;
if> end if;
```

However, there is a quicker way to perform this task, and that is to use a `select` command.

```
> s := (a gt 0) select 1 else -1;
```

Thus the syntax is

```
      (variable):= (condition) select (value if true) else (value if false)
```

A nifty trick with these `select` functions is that they can be nested, if there are several possibilities. For example, if we allowed `a` to be zero in the above example, there would be three possibilities for `s`.

```
s := (a gt 0) select 1 else ((a eq 0) select 0 else -1);
```

[Of course, another way is to use the command `Sign`, which is done by `s:=Sign(a)`, but the example still stands. We will study functions involving numbers in greater detail in Chapter 4.]

We come to the second special type of if statement, which deals with the situation above with congruences modulo 4. This is the `case` command. The syntax is given in the example below.

```
> x := 5;
> case Sign(x):
>  when 1:
>   "x is positive";
>  when 0:
>   "x is zero";
>  when -1:
>   "x is negative";
> end case;
x is positive
```

[This example has been indented to show the structure of the code. With the `case` commands, Magma fails to inform you that you are inside a loop in the way it normally does.] The idea is that some variable (in this case `Sign(x)`) has a finite list of possibilities, and for each of them we detail what Magma should do.

Having dealt with conditions, we will deal briefly with Boolean operators. These are fairly obvious, and we will spend very little time on them. There are commands `not`, `or`, `and`, and `xor`, and these are used in the obvious way. If brackets are not provided, then a sequence of booleans is assumed to be right-associative, as the example below demonstrates.

```
> x:=5;
> x eq 5 or x eq 4 and x eq 2;
true
```

Exercise 1.5 Write a command that takes a variable x and produces $|x|^3$ using conditions. Do this three times, using `if`, `select` and `case`.

Extended Exercise 1.6 In this exercise we will construct a Lotto machine! The Lotto (aka National Lottery) involves the player choosing 6 numbers between 1 and 49, and then the Lotto machine picks six, and the prize is determined by the quantity of matching numbers: no prize for 0, 1 or 2 matching balls; £10 for three; about £65 for four; about £2000 for five, and the Jackpot for all six. (We are ignoring the bonus ball.) Write a program that takes six numbers as inputs, checks that they are valid Lotto numbers, draws six Lotto numbers (recall the `Random` function) and tells the lucky (or more likely unlucky) player how much they have won. (You might have to run this more than fifty times to win something...)

Chapter 2

Iteration and Function Building

The idea of this chapter is to finish the rest of the general skills you need to start building your own Magma programs. In the succeeding chapters, we will look at combinatorics and graph theory, at number theory and linear algebra, at group theory, and at representation theory. This chapter will first look at how to iterate over finite sets, then move onto how to deal with strings (of letters, not the physics ones or balls thereof) and how to control what gets sent to the terminal, about saving to and loading from text files, and about creating your own functions.

2.1 Iteration

The main power of Magma lies not in the fact that it can produce sets, but that it can iterate over them. For instance, consider the following example.

```
> X:={1..5};
> for i in X do i^2; end for;
1
4
9
16
25
```

The for loop is one of the most fundamental of all operations in Magma. Broadly speaking, Magma has the ability to iterate over any finite structure that has an underlying collection of objects, such as a set, multiset, group, finite field, and so on.

The syntax of a for loop is not difficult: it is of the form

```
> for i in A do
for> COMMAND;
for> COMMAND;
for> end for;
```

Here, `A` is a collection of objects over which we want to iterate, and `i` is the variable that the elements of `A` will be assigned during that loop. Notice the lack of a semicolon after the `do`.

As an example, let us calculate the congruence classes modulo 7 for $x^3 + 6x^2 + 15$, as x ranges between the integers 1 and 20.

```
> Ans:=[];
> for i in [1..20] do
for> Ans:=Append(Ans,(i^3+6*i^2+15) mod 7);
for> end for;
> Ans;
[ 1, 5, 5, 0, 3, 6, 1, 1, 5, 5, 0, 3, 6, 1, 1, 5, 5, 0, 3, 6 ]
```

We could have simply outputted them like in the previous example, but that would have taken up too much room.

There are a few other loops, which are similar in style to a for loop, but which have slightly different uses. The first of these is a while loop. These can be used when there is no object over which we can iterate. The idea is to keep testing a condition over and over again, and keep looping until the condition is satisfied. For example, suppose that we have a number n, and want to know the next multiple of 5 bigger than n. There are other ways of doing this, but one way is to use a while loop.

```
> n:=453;
> while((n mod 5) ne 0) do
while> n+:=1;
while> end while;
> n;
455
```

There are some situations where while and for loops can be used interchangeably, and some situations where one is much more useful than the other. The for command is used generally when you want to iterate over a specified object, or run a command a specific number of times. The while command, on the other hand, is much better for indefinite looping, for example if a particular function only has a probability of producing a useful answer, and so you might want to run it an indefinite (but hopefully finite!) number of times.

A significant problem with a while loop is that if the condition is satisfied before the loop is reached in the program, the while loop *is never run*, not even once. Sometimes this can be annoying, and rather than specify the contents of the loop to run once before the while loop, we use a different command, called the `repeat` function.

The repeat function is the same as the while function, except that the particular condition is tested at the *end* of the loop, not at the beginning. This means that regardless of the condition, the loop will always run at least once. This is particularly useful when the condition of the loop depends on the output of the loop. We give a demonstrating example below.

```
> X:={1..1000};
> repeat a:=Random(X);
repeat> bool:=IsPrime(a);
repeat> until bool;
> a;
911
```

[The function of the command `IsPrime` is presumably obvious.] Here we see that the command structure is '`repeat (commands) until (condition)`'. In this case, whether we exit from the loop depends on the output of the loop the first time round, and so a while loop would have to be preceded with the command `bool:=false`, or taking a random element of `X` first. The loop above can be shortened somewhat by using the output of `IsPrime` as the condition, as below.

```
> X:={1..1000};
> repeat a:=Random(X); until IsPrime(a);
> a;
887
```

The last concept in this section is a slightly tricky one: that of breaks and continues. These exist in programming languages as well, so if you know one of those, then this section will be easy. Otherwise, the idea of a break command is to terminate the loop early before the condition specified by the while or repeat loop is satisfied, or before the for loop has finished.

We will illustrate this with the last example. In fact, we were lucky before, because the set `X` actually had a prime in it. The worry for us (and for any programmers using while loops!) is that although the for loop definitely will terminate, as it's looping over a finite object, the while and repeat loops might *never terminate*, because the conditions for their termination cannot occur.

```
> X:={2*x:x in {2..10}};
> repeat a:=Random(X); until IsPrime(a);

[Interrupted]
```

Here we see what happens when you enter an infinite loop: after a certain amount of time, you get bored and terminate the procedure. (To make Magma interrupt what it's doing, press the 'Control' key and the 'C' key simultaneously. Depending on what Magma is doing, this will stop at the next command, or maybe before it has finished with its current command. Unfortunately, if, for example, you have asked Magma to decompose a 100,000-dimensional module into its indecomposable summands, this command will last a long time, and trying to interrupt it doesn't work. The 'nuclear option' is to press Control and C twice within half a second, which terminates Magma itself, and you lose everything. This is annoying.)

In this case, we can get out of our problems, by using a 'break' command.

```
> X:={2*x:x in {2..100}};
> for i in [1..3*#X] do
for> a:=Random(X);
for> if(IsPrime(a)) then a; break; end if;
for> end for;
> X:={1..10000};
> for i in [1..3*#X] do
for> a:=Random(X);
for> if(IsPrime(a)) then a; break; end if;
for> end for;
5927
```

[Notice that there was no output from the first set of commands.] The command 'break' exits out of the nearest iterative loop (i.e., for, while, and repeat). Since there is only one loop here, it exits out of this one. We chose $3|X|$ for the number of times we tested an element from the set `X` because by then there is a pretty good chance that there isn't a prime in `X` at all, and if we are determined to check, we can loop over all elements of `X` using a for command.

The break command can also be used to quickly get out of a situation in case a weird answer has been found, or some unlikely condition has been met. One of the easiest loops to code, and possibly quite dangerous in terms of having an infinite loop, is to use the command `while(true)`, with a `break` command inside the loop as the exit point. This can be a useful trick if you have several ways of getting out of the loop, and you have code in between these exits, say. We will see an example of this when we code the dice game of Craps in the exercises. (Rules to Craps are included!)

The break command can be used to exit one for loop, but what if you need to exit more than one loop at the one time? The break command can exit from an outer for loop, by including the name of the variable used in the for loop after the break.

```
> p := 10037;
> for x in [1..100] do
for> for y in [1..100] do
for|for> if x^2 + y^2 eq p then
for|for|if> print x, y;
for|for|if> break;
for|for|if> end if;
for|for> end for;
```

```
for> end for;
46 89
89 46
> for x in [1..100] do
for> for y in [1..100] do
for|for> if x^2 + y^2 eq p then
for|for|if> print x, y;
for|for|if> break x;
for|for|if> end if;
for|for> end for;
for> end for;
46 89
```

[This example was shamelessly ripped from the Magma help files, and will be replaced with my own example at some point.]

The final command in this section is very similar in nature to break, but very different in outcome. The continue command can be put in the same position as a break statement, and its effect is to go back to the start of the loop, without executing the other commands left in this iteration.

Another example, again stolen from the Magma help file, is given below. Suppose (obviously) that we needed to know the product of all composite numbers between 2 and 50. This is easiest achieved as follows.

```
> product := 1;
> for i in [2..50] do
for> if IsPrime(i) then
for|if> continue;
for|if> end if;
for> product *:= i;
for> end for;
> print product;
49462674552307032732500869107626803200000000000
```

Exercise 2.1 Using your new-found skills at iteration, make your program for a Lottery machine much nicer. Wrap the entire thing in an outer for loop that, when you specify a set of six numbers and an integer n, plays the Lottery n times with those numbers, and outputs either the number of times you win each prize or your earnings (whichever you want!).

Exercise 2.2 Here we will make a machine that will play Craps for us. The rules to Craps are basically as follows. The shooter (person who throws) has two dice, and throws them the first time.

If he (it's normally a he) throws a 7 or 11, he wins immediately. If he throws a 2, 3 or 12, he loses immediately. If he throws anything else, he will have to throw again, and the total on the dice in this first throw is the 'point'. The shooter continues to roll the dice until either a 7 occurs (and he loses) or his point occurs (and he wins). Encode a machine that will play this game. The more adventurous of you can play this game, say 10000 times, and see how often they win to lose. (The house edge in this game is very small, at around 1.5%.)

2.2 Procedures Versus Functions

This short section will not really introduce any more commands, but will hopefully clear up some potential problems that you might have with what commands like `Maximum(X)` actually do. Suppose that we have executed the command `x:=5`, and we want to increase `x` by 1. From the very first section, we know that there are two different ways to do it:

```
> x:=5;
> x:=x+1; x;
6
> x+:=1; x;
7
```

At the time it was mentioned that the second method was slightly faster than the first. The first method uses a *function*, and the second method uses a *procedure*. To understand the difference, notice what happens when we type `x+1` into Magma.

```
> x:=5;
> x+1;
6
```

The command `x+1` produces an *output*, namely 6, that Magma displays on the screen. The assignment operator `:=` takes the output and makes the variable `x` point to it. This has the effect of incrementing the variable by 1. There is, however, another way.

```
> x:=5;
> x+:=1;
> x;
6
```

In the command `x+:=1`, no output at all was produced. The variable `x` was increased by 1 internally. Thus there are two different types of command happening here: a *function*, which takes inputs and produces outputs; and a procedure, which takes inputs and alters them.

We will illustrate the difference with a more explicit example. A few sections ago, we considered the commands `Include` and `Exclude`. These added and removed an element from a set. They come in two flavours: functional and procedural. To make `Include` a procedure, we include a tilde: this example demonstrates.

```
> set:={3,7,4,1,0};
> Include(set,6);
{ 0, 1, 3, 4, 6, 7 }
> set;
{ 0, 1, 3, 4, 7 }
> Include(~set,6);
> set;
{ 0, 1, 3, 4, 6, 7 }
```

With the second command, the actual set given by the variable `set` has changed, whereas to get that effect without using a procedure, we would have to use the command

```
> set:=Include(set,6);
```

As with the previous example, the procedure is slightly quicker.

In Magma, procedures tend to exist when you are using a command to alter an object, such as by applying addition, or appending an element to a set. They do not exist for all functions, and if you are unsure, it is recommended to look at the Magma on-line help for more information.

We will give another final example in this section.

```
> set:={ 0, 1, 3, 4, 6, 7 };
> set2:={ -3, 1, 4, 6 };
> set join:=set2;
> set;
{ -3, 0, 1, 3, 4, 6, 7 }
```

2.3 Writing Functions and Procedures

Now comes the time to produce our own functions and procedures that can be integrated seamlessly into the Magma program itself. The idea is to create a function, say `MyFunction`, which can be called just like `Maximum` or `IsPrime`. The syntax to do this is easy. (There are in fact two ways to do this, but we will use only one here. See the help pages for more details.)

```
> function Fib(n)
function> if(n le 0) then return 0; end if;
function> if(n le 2) then return 1; end if;
function> return Fib(n-1)+Fib(n-2);
function> end function;
> Fib(5);
```

```
5
> Fib(8);
21
```

Thus the syntax for a function is to write `function FUNCTIONNAME(PARAMETERS)` at the start, then give the commands, with the output preceded by a `return` command. It is closed with an `end function`. Easy.

Suppose that we want to return multiple outputs, not just the one, just like functions we have seen before. This is possible using comma separation. Suppose that, given a group G of order n, we want to know the order and index of a Sylow p-subgroup of G. Thus we need to write n as $p^a m$, where $p \nmid m$. Consider the code below.

```
> function PPart(n,p);
function> if(not(IsPrime(p))) then return "p is not a prime"; end if;
function> a:=0;
function> while(n mod p eq 0) do
function|while> a+:=1;
function|while> n:=n/p;
function|while> end while;
function> return n,p^a;
function> end function;
```

Before typing this into Magma, you should try to predict what it will do. Suppose that we fed it $n = 60$ and $p = 2$; it should return `15 4`, right? Actually no: it will throw up an error, because of a problem we ran into in the previous chapter.

```
> function PPart(n,p);
function> if(not(IsPrime(p))) then return "p is not a prime"; end if;
function> a:=0;
function> while(n mod p eq 0) do
function|while> a+:=1;
function|while> n div:=p;
function|while> end while;
function> return n,p^a;
function> end function;
> PPart(60,2);
15 4
```

This example was placed here as a continual reminder that division makes rationals, even when you have checked that it shouldn't!

There is one more thing that we need to think about before looking at procedures, and then you can be let loose on a couple of exercises. What happens if you want a function that returns different numbers of outputs depending on the inputs? This requires the use of an underscore. [Again, we steal this from the Magma on-line help.]

```
> function TestOdd(x)
function> if IsOdd(x) then
function|if> return true, x;
function|if> else
function|if> return false, _;
function|if> end if;
function> end function;
>
> TestOdd(1);
true 1
> TestOdd(2);
false
> a, b := TestOdd(1);
> a;
true
> b;
1
> a, b := TestOdd(2);
> a;
false
> b;

>> b;
   ^
User error: Identifier 'b' has not been assigned
```

Another, more useful example of a function is the following, which calculates random Poisson variables.

```
function RandomPoisson(x)
k:=0;
max_k:=1000;
p:= Random([1..10^5])/10^5;
P:= Exp(-x);
sum:=P;
if(sum ge p) then return 0; end if;
for k in [1..max_k] do
```

```
    P*:=x/k;
    sum+:=P;
    if (sum ge p) then return k; end if;
end for;

return k;

end function;
```

The idea of this function is to construct the smallest k such that

$$\sum_{i=1}^{k} \frac{e^{-x} x^i}{i!} \geqslant p,$$

where p is a uniform random variable. (Unhelpfully, Magma only produces random variables that are uniformly distributed. This function may therefore come in handy!)

A procedure is similar to a function, in that it takes inputs. However, it differs from a function in that it changes the inputs, rather than produces outputs.

We will construct a procedure that performs the calculation `x*:=y` again, to demonstrate the syntax.

```
> procedure Multiplies(~x,y)
procedure> x:=x*y;
procedure> end procedure;
> x:=2; y:=3;
> Multiplies(~x,y);
> x;
6
```

Each variable that needs to be altered is given a tilde in front.

Exercise 2.3 We shall complete our gambling den by turning our bespoke machines into functions. Create the functions `Lottery` and `Craps`: the first should take as an input a set of six numbers and output the money won, and the second should take no inputs and return 1 or 0 depending on whether you win or lose.

Exercise 2.4 Test out the statement, given earlier, that the function `Random(X)` produces a uniform probability of choosing each element from the set X. Compare it to the distribution got from the function `RandomPoisson` described above.

Extended Exercise 2.5 Create a function that decides whether two line segments intersect. Each line segment will be given in the form

```
[[x,y],[a,b]]
```

where `[x,y]` is the start point and `[a,b]` is the end point of the line segment. The elements `x`, `y`, `a`, and `b` are real numbers from some real field `R`. [This function is not difficult, but it may be helpful to do some of the elementary geometry on a piece of paper first and input the equations into Magma afterwards.]

2.4 Saving and Loading

Saving and loading in Magma is extremely easy! (At least one thing is...) Suppose that you have a file called 'magmastuff.txt', and it is loaded in your home directory. Then to read all of the commands into Magma as if you typed them in, you simply type

```
> load "magmastuff.txt";
```

For example, suppose that you have a text file saying '`a:=1; b:=2;`', and this file's path was 'magmaprogs/variables' (i.e., it was a file called 'variables' in a directory called 'magmaprogs'). Then you could do this:

```
Magma V2.13-15    Thu Jan 24 2008 23:09:56 on felix-the-cat [Seed = 332222129]
Type ? for help.  Type <Ctrl>-D to quit.
> load "magmaprogs/variables";
> a+b;
3
```

This is particularly useful when you have a library of functions, or some data like a representation of a group by matrices to load in.

It is also possible to save the internal state of Magma, using the '`save`' command. This is not highly recommended if you have been doing something complicated, because Magma might produce a rather large file. This is particularly unhelpful in the Mathematical Institute's case, where everyone has a limited amount of hard drive space. The best way to accomplish this is to get Magma to output the information and transfer it to a text file, so that you can read it back in with a `load` command.

```
> save "magmastate1";
```

A few days later, one can get back this state with

```
> restore "magmastate1";
```

When you restore a state, it destroys everything you were doing beforehand in that Magma session. So don't come crying to me if you lose all your work this way.

2.5 Outputting to Screens and Files

We have already seen a few ways of outputting to a screen.

```
> a:=3;
> a;
3
> print "Hello!!!";
Hello!!!
```

The first time Magma was outputting a variable because you asked it what it was: the second time, we used a string of letters.

```
> str:= "Hello";
> Type(str);
MonStgElt
```

Things in speech marks are *strings*, and are of type `MonStgElt` in Magma. Strings can be accessed like sequences, and some of the same functions and procedures can be applied to them.

```
> X:="Hello";
> Y:="world";
> X cat " " cat Y;
Hello world
> Prune(X);
Hell
> X[3];
l
> #X;
5
> Index(X,"e");
2
```

If you want to concatenate a sequence of strings to together, or separate a string into its constituent characters, then you should be happy because these can both be accomplished.

```
> X:="Hello";
> Z:=["Hello", " ", "world"];
> &cat Z;
Hello world
> ElementToSequence(X);
[ H, e, l, l, o ]
```

The `& cat Z` function takes a sequence `Z` of strings and concatenates them together. In fact, the same syntax can be used to applying a binary operation to any sequence of objects.

```
> A:=[2,3,4,1,5];
> &+ A;
15
```

This can be a useful operation.

Passing between strings and integers is also a useful skill, particularly passing from integers to strings.

```
> a:="342";
> IsEven(a);

>> IsEven(a);
         ^

Runtime error in 'IsEven': Bad argument types
Argument types given: MonStgElt

> IsEven(StringToInteger(a));
true
> x:=5;
> print "The value of x is " cat x;

>> print "The value of x is " cat x;
                               ^

Runtime error in 'cat': Bad argument types
Argument types given: MonStgElt, RngIntElt

> print "The value of x is " cat IntegerToString(x);
The value of x is 5
```

Equality can also be tested for strings. The following amazing example comes from Magma's on-line help.

```
> "73" * "9" * "42" eq "7" * "3942";
true
> 73 * 9 * 42 eq 7 * 3942;
true
```

In the first case, the `*` operation acts like `cat` does, and in the second it is the usual multiplication of numbers. Strings can also be compared according to the lexicographic ordering, with `lt`, `le`, `gt`, and `ge` commands. Finally, the commands `in` and `notin` test whether the one string is a substring of the other. All of these commands are case-sensitive, and `in` requires the substring to be contiguous inside the larger string.

```
> "ag" in "Magma";
true
> "ag" in "Magma";
false
> "am" in "Magma";
false
```

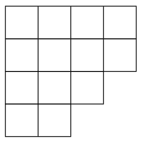
Having dealt with strings and outputting things to the screen, we will now describe the simple method used in diverting output from the screen to a file. This is particularly useful when you have a large amount of output (for example, a huge matrix or two in representation theory) and want it to go directly to a file.

```
> x:=5;
> SetOutputFile("myoutput.txt");
> x; 2*x; print "Hello!!";
> UnsetOutputFile();
> x; 2*x; print "Hello!!";
5
10
Hello!!
```

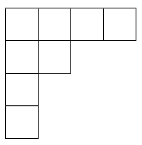
The `SetOutputFile` command diverts **all** output from Magma to the file specified (which need not exist, in which case Magma creates it for you), until you use the command `UnsetOutputFile`.

Exercise 2.6 Construct a sequence containing the numbers 1 to 20 using the `X:=[1..20]` command. Now get Magma to output it by typing `X;`, and you will see that it outputs `[ 1 .. 20 ]`. Using strings and concatenation, make Magma output the sequence `X` one number at a time, as it would be inputted (i.e., as `[1,2,` and so on.) Write your code so that if `X` were changed to a different sequence of numbers (with a possibly different length) it would still work.

Extended Exercise 2.7 In this exercise, we will produce a Magma program to construct a t-core of a partition. Recall that if λ is a partition, then it can be thought of as a tableau, for example $(4, 4, 3, 2)$ can be thought of as



A *rim t-hook* of λ is a connected selection of t boxes on the outer edge (i.e., the bottom-right edge) of λ such that when they are removed, the resulting diagram is still a partition. For example, a rim 5-hook of the above partition can be seen, and upon removing it we are left with the partition



A partition λ is a t-core if λ possesses no rim t-hooks. We will write two programs: the first will produce the t-core of a given partition, and the second will compute the number of partitions of n that are t-cores.

Recall also that the *hook length* of a box in a partition is the sum of the number of boxes below it, the number of boxes to the right of it, and the box itself. The hook lengths of the partition above are given now.

7	6	4	2
6	5	2	1
4	3	1	
2	1		

The first column hook lengths are the hook lengths given in the first column, and form a subset of the positive integers. Every subset of the integers corresponds to some partition and vice versa.

We will now give some of the commands that Magma has built into it for dealing with partitions. Partitions are considered to be weakly decreasing sequences of numbers, such as `[4,4,3,2]` above. A sequence of integers can be tested to see if it is a partition with the command `IsPartition`. (It throws an error if the argument isn't at least a sequence of integers.) However, `IsPartition` allows zeros to be in the partition, and so if you need to reject such sequences, either build a new function to discount this case or manually check for the presence of zeros.

The sequence of all partitions of size n can be produced with the command `Partitions(n)`, whereas if you merely want know how many partitions there are, the command `NumberOfPartitions(n)` will do. Finally, to get the hook length of the box in the ith row and the jth column of the partition `X`, you write `HookLength(X,i,j)`.

(i) Construct a program such that, when given a partition as a weakly decreasing sequence of integers such as `[4,4,3,2]`, produces a sequence of its first column hook lengths (strictly decreasing).

(ii) Construct a program such that, when given a strictly decreasing sequence of positive integers (thought of as first column hook lengths), constructs the corresponding partition.

(iii) (Difficult) Construct a program that removes a rim t-hook from a partition. If a rim t-hook can be found, return the partition with this removed, together with `true`. If no rim t-hook can be found, the original partition should be returned along with `false` (Hint: One method is to move along the boxes of the rim checking whether removing t boxes along the rim from this box results in a partition.) Those people who know about James' abacus have the extra task of encoding that method in as well.

(iv) Write a program that produces the t-core of a partition from a given partition, using the previous function.

(v) Write a program that, given numbers n and t, counts the number of partitions such that no rim t-hooks can be removed, exactly one can, two can consecutively, and so on. (The sum of all these numbers should be the number of partitions of n...)

Chapter 3

Combinatorics and Graph Theory

In the extended exercise of the last chapter, we saw some of the combinatorial tools available in Magma for dealing with partitions. Magma can be used to perform a wide range of combinatorial operations, and has many of the more common ones built in.

In this chapter, we will see Magma's capabilities with general combinatorics such as the binomial distribution, with the combinatorics of subsets, with graph theory, and with partitions. Untouched, but implemented in Magma, are its functions concerning design theory, networks and Hadamard matrices. It can also be used to study symmetric functions.

3.1 General Combinatorics

One of the most obvious things that anyone working in combinatorics needs is the binomial function. This, together with the factorial and multinomial function are given below.

```
> Binomial(7,4);
35
> Factorial(6);
720
> Multinomial(10,[3,4,3]);
4200
```

Generalizing the binomial coefficients, we have Stirling numbers of the first and second kind. The Stirling numbers of the first kind are signed, so that we have the examples below.

```
> StirlingFirst(10,5);
-269325
> StirlingSecond(10,5);
42525
> StirlingFirst(10,6);
63273
```

```
> StirlingSecond(10,6);
22827
```

The Fibonacci, Lucas, and Catalan numbers are all hard-encoded into Magma as well.

```
> Fibonacci(10);
55
> Lucas(10);
123
> Catalan(10);
16796
```

The two numbers associated with partitions, the Bell numbers (numbers of set partitions) and the number of partitions of an integer, are also calculable in Magma.

```
> NumberOfPartitions(10);
42
> Bell(10);
115975
```

As well as constructing the Bell number, Magma can also construct subsets of a given set.

```
> S:={1,2,3,4};
> Subsets(S);
{
    {},
    { 4 },
    { 2, 3 },
    { 2, 3, 4 },
    { 3, 4 },
    { 1, 3 },
    { 3 },
    { 1 },
    { 1, 4 },
    { 1, 2, 3, 4 },
    { 1, 2, 3 },
    { 2 },
    { 1, 2 },
    { 1, 2, 4 },
    { 2, 4 },
    { 1, 3, 4 }
}
```

If you just want those of cardinality k, you can use `Subsets(S,k)`. If you want to be able to choose k objects from S *with replacement*, then you can use `Multisets`.

```
> Multisets({1,2,3},2);
{
    {* 1^^2 *},
    {* 1, 3 *},
    {* 2^^2 *},
    {* 3^^2 *},
    {* 2, 3 *},
    {* 1, 2 *}
}
```

Finally, one may actually create sequences of these, which can be thought of as permutations.

```
> Permutations({1,2,3});
{
    [ 2, 1, 3 ],
    [ 1, 3, 2 ],
    [ 1, 2, 3 ],
    [ 2, 3, 1 ],
    [ 3, 2, 1 ],
    [ 3, 1, 2 ]
}
> Permutations({1,2,3},2);
{
    [ 1, 3 ],
    [ 1, 2 ],
    [ 2, 3 ],
    [ 3, 2 ],
    [ 3, 1 ],
    [ 2, 1 ]
}
```

Exercise 3.1 Construct a function that emulates the recurrence relation $x_{n+1} = x_n - (x_n^2)/n$, where $x_0 = 1/2$. (Hint: One may call a function from within itself.)

Exercise 3.2 The Delannoy numbers enumerate the number of paths on a square grid of size n that start from the bottom-left and end at the top-right, moving only north, east, and north-east one square at a time. They are given by the recurrence relation

$$n\,a_n = 3 \cdot (2n-1)a_{n-1} - (n-1) \cdot a_{n-2}$$

where $a_{-1} = a_0 = 1$. Encode this recurrence relation into Magma, and use to it calculate a_{20}.

3.2 Partition Theory

In the exercises of the previous chapter, we saw some of the methods employable when dealing with partitions. Here, we will flesh out this list by giving the reader many more of the commands available. We have supplemented these commands with a couple of custom-built functions of our own, particularly useful when considering the representation theory of the symmetric group. (There are tableaux-based methods that will not be discussed here for lack of space, and the fact that it is a rather specialized topic. The Magma on-line help has more details.)

We have seen the function `NumberOfPartitions`, which uses Euler's pentagonal formula to easily calculate the number of partitions. If one actually requires a sequence of all partitions, this is made possible using `Partitions`. We illustrate its use here, along with some related functions.

```
> Partitions(6);
[
    [ 6 ],
    [ 5, 1 ],
    [ 4, 2 ],
    [ 4, 1, 1 ],
    [ 3, 3 ],
    [ 3, 2, 1 ],
    [ 3, 1, 1, 1 ],
    [ 2, 2, 2 ],
    [ 2, 2, 1, 1 ],
    [ 2, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1 ]
]
> Partitions(6,3);
[
    [ 4, 1, 1 ],
    [ 3, 2, 1 ],
    [ 2, 2, 2 ]
]
> RestrictedPartitions(6,{1,2});
[
    [ 2, 2, 2 ],
    [ 2, 2, 1, 1 ],
    [ 2, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1 ]
]
> RestrictedPartitions(8,4,{1,2,3});
[
    [ 3, 3, 1, 1 ],
    [ 3, 2, 2, 1 ],
    [ 2, 2, 2, 2 ]
]
```

The function `Partitions(n,k)` returns all partitions of n into k parts, and `RestrictedPartitions(n,S)` returns all partitions of n using only parts from the set S. The final command returns all partitions of n into k parts each coming from S.

Whether a sequence of integers is a partition or not can be tested with `IsPartition`, which doesn't appreciate being given anything other than a sequence of integers.

```
> IsPartition([4,4,2]);
true
> IsPartition([4,4,2,3]);
false
> IsPartition([4,4,2,3.0]);

>> IsPartition([4,4,2,3.0]);
              ^
Runtime error in 'IsPartition': Argument should be a sequence of integers
> IsPartition([0,0]);
true
```

To get a partition at random without constructing all partitions beforehand, use the command `RandomPartition`. The collection of all partitions is returned using the lexicographic ordering. The position of a partition in this ordering, which starts from 0, can be obtained using `IndexOfPartition`, And the weight of a partition—that is, of which integer it is a partition—can be obtained using `Weight`.

```
> RandomPartition(10);
[ 3, 3, 2, 2 ]
> Weight([4,4,2]);
10
> Partitions(6);
[
    [ 6 ],
    [ 5, 1 ],
    [ 4, 2 ],
    [ 4, 1, 1 ],
    [ 3, 3 ],
    [ 3, 2, 1 ],
    [ 3, 1, 1, 1 ],
    [ 2, 2, 2 ],
    [ 2, 2, 1, 1 ],
    [ 2, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1 ]
]
> IndexOfPartition([3,1,1,1]);
6
> IndexOfPartition([6]);
0
```

These are the commands that deal with partitions. There are many more commands dealing with tableaux, implementing for example the *jeu de taquin*, the Schensted algorithm, and Knuth equivalence, to name but a few of the functions. We will concern ourselves here with constructing a few more functions dealing with taking cores of partitions, as we did in the previous chapter's extended exercise.

```
function FirstColumnHookLengths(X)

if(#X eq 0) then return X; end if;
if(not(IsPartition(X)) or X[#X] eq 0) then print "This is not a partition";
  return [0];
end if;

Ans:=[];
for i in [1..#X] do
  Append(~Ans,HookLength(X,i,1));
end for;

return Ans;
end function;
```

After inputting this into Magma, we may test it out.

```
> FirstColumnHookLengths([4,4,2]);
[ 6, 5, 2 ]
```

Having created a function to extract the first column hook lengths, we should also construct a function to turn the first column hook lengths back into a partition.

```
function UnderlyingPartition(X)
if(#X eq 0) then return X; end if;

for i in [1..#X-1] do if(X[i]-X[i+1] le 0) then
    print "Not the first column hook lengths of a partition";
    return [0];
end if; end for;

Ans:=[];
for i in [1..#X] do Append(~Ans,X[i]+i-#X); end for;

return Ans;
end function;
```

After inputting this into Magma, we may test it out.

```
> FirstColumnHookLengths([4,4,2]);
[ 6, 5, 2 ]
> UnderlyingPartition([6,5,2]);
[ 4, 4, 2 ]
```

Having constructed these functions, we can now implement James' abacus to construct the t-core of a partition. We will not explain James' abacus here, and this code is only given here so that it may be used in calculating t-cores.

```
function TCore(X,t);
if(#X eq 0) then return X; end if;

if(not(IsPartition(X))) then print "This is not a partition";
  return [0];
end if;

Y:=SequenceToSet(FirstColumnHookLengths(X));

repeat
  nochange:=true;
  for i in Y do
    if(i-t ge 0 and i-t notin Y) then
```

```
      Y:=Include(Exclude(Y,i),i-t);
      nochange:=false;
    end if;
  end for;
until nochange;

if(0 in Y) then
  i:=0;
  repeat
    Exclude(~Y,i);
    i+:=1;
  until i notin Y;
  Y:={j-i:j in Y};
end if;

Y:=Reverse(SetToSequence(Y));
return UnderlyingPartition(Y);
end function;
```

Using this method, it is easy to compute the t-blocks of the symmetric groups, which is the main representation-theoretic reason for considering t-cores.

Exercise 3.3 Here we will confirm for some small values Euler's famous theorem that the number of partitions into distinct odd parts is equal to the number of self-conjugate partitions. Create the functions `OddDistinctPartitions(n)` and `SelfConjugatePartitions(n)` to construct the relevant sequences, and confirm that they have the same size for n from 1 to 30. It may be helpful to know that the function `ConjugatePartition` returns the conjugate partition. (You may use the function `Partitions(n)` to begin with.)

Exercise 3.4 When dealing with partitions of integers larger than around 50, constructing the set of all partitions beforehand is rather cumbersome, and the computer will eventually run out of memory doing this. Create a function that when given a particular partition, outputs the next partition lexicographically. (The lexicographic ordering is that used in the Magma procedure, so if you don't know what this is, you can use the `Partitions` function to observe its behaviour.)

Exercise 3.5 A multipartition of n into k parts is an ordered collection of k partitions, the sum of whose weights is n itself. (Any of the k partitions may be empty in this case, although this is not always true in examples.) Construct a function that returns all multipartitions of n into exactly k partitions.

3.3 Graph Theory

In mathematics, a graph is a set of vertices and a set of pairs of vertices. This is a method of constructing graphs in Magma as well. This is a construction of the well-known Petersen graph.

```
> P := Graph< 10 | {1,2},{1,5},{1,6},{2,3},{2,7},
> {3,4},{3,8},{4,5},{4,9},{5,10},
> {6,8},{6,9},{7,9},{7,10},{8,10}>;
```

Magma has some very powerful graph-theoretic algorithms, including a linear-time algorithm for proving whether a given graph is planar. It can also be used to produce the automorphism group of a graph, and contains a database of strongly regular graphs, for example. We will see some of Magma's capability in this direction.

The construction of the graph above uses one of the methods available in Magma. The previous sentence would imply that there are other methods available. Another way to define a graph is by defining the neighbours of each vertex.

```
> G:=Graph<5|[{2,3,4},{1,3,4},{1,2,4,5},{1,2,3,5},{3,4}]>;
```

The graph above is famous: it is the house graph that people try to draw without taking the pencil off the paper.

There are two obvious ways to store a graph: the first is by an adjacency matrix, and this is the method favoured by Magma. Another method is to merely store an adjacency list: this is much more preferable with sparse graphs, where the adjacency matrix might be rather large and unwieldy. If this is the method you would like, you can alter the constructor, as follows.

```
> G:=Graph<5|[{2,3,4},{1,3,4},{1,2,4,5},{1,2,3,5},{3,4}] : SparseRep := true>;
```

Directed graphs can also be constructed in Magma. To do so, we use the command `Digraph`, as follows.

```
> D:=Digraph<5|[1,2],[2,3],[3,4],[4,5],[5,1]>;
```

This then constructs the wheel graph, with an arrow that goes from vertex i to $i+1$. Sparse representations are also allowed for digraphs, if you are intending on producing very big digraphs.

Certain special kinds of graphs are built into Magma: complete graphs, polygonal graphs, multipartite graphs, and so on.

```
> G1:=CompleteGraph(10);
> G2:=EmptyGraph(10);
> G3:=PathGraph(10);
> G4:=PolygonGraph(10);
> G5:=KCubeGraph(3);
> #Edges(G1); #Edges(G2); #Edges(G3); #Edges(G4); #Edges(G5);
```

```
45
0
9
10
12
```

The complete and empty graphs are obvious; the path graph is a tree that consists of a single line; the polygon graph consists of one cycle, and the final graph `KCubeGraph(n)` is the graph of the n-dimensional cube.

There are a couple of particular graphs already in Magma.

```
> G:=ClebschGraph();
> Diameter(G);
2
> H:=ShrikhandeGraph();
> #Vertices(H);
16
> J:=GewirtzGraph();
> #Edges(J);
280
```

As well as these, we can construct bipartite and multipartite graphs.

```
> G:=BipartiteGraph(2,3); G;
Graph
Vertex  Neighbours

1       3 4 5 ;
2       3 4 5 ;
3       1 2 ;
4       1 2 ;
5       1 2 ;
> H:=MultipartiteGraph([2,3,1]); H;
Graph
Vertex  Neighbours

1       3 4 5 6 ;
2       3 4 5 6 ;
3       1 2 6 ;
4       1 2 6 ;
5       1 2 6 ;
```

```
6       1 2 3 4 5 ;
```

Random graphs can be constructed in Magma as well.

```
> G:=RandomGraph(1000,0.3);
> #Edges(G);
149807
> Binomial(1000,2)*0.3;
149850.000000000000000000000000
> H:=RandomTree(100);
> Degree(Random(Vertices(H)));
3
> Degree(Random(Vertices(H)));
2
```

One particularly powerful method to produce graphs is to construct an empty graph, and then add edges one at a time.

```
> G:=Graph<10|>;
> for i in [1..5] do AddEdge(~G,i,2*i); end for;
> G;
Graph
Vertex  Neighbours


1       2 ;
2       1 4 ;
3       6 ;
4       2 8 ;
5       10 ;
6       3 ;
7       ;
8       4 ;
9       ;
10      5 ;
```

We began by creating an empty graph, and then added a series of edges according to some rule. Edges are added using the `AddEdge` procedure, which takes the graph, start vertex and end vertex as arguments. It is also possible to add vertices. The example above continues.

```
> AddVertices(~G,2);
> G;
Graph
```

```
Vertex  Neighbours

1       2 ;
2       1 4 ;
3       6 ;
4       2 8 ;
5       10 ;
6       3 ;
7       ;
8       4 ;
9       ;
10      5 ;
11      ;
12      ;
```

This is not an exhaustive list of all graph constructions available in Magma, but these are some of the more important functions. A complete list of all graph-related functions in Magma can of course be found with the on-line help.

For directed graphs, there are the standard constructors `CompleteDigraph`, `EmptyDigraph` and `RandomDigraph`.

Now we'll move on to working with graphs. For this, we will look at the vertices and edges of a graph. There are two ways of accessing the vertices (and edges) of a graph: by using `Vertices(G)`, and by using `VertexSet(G)`.

```
> G:=Graph<6|{1,2},{1,3},{1,4},{2,3},{2,5},{3,6},{5,6}>;
> V:=VertexSet(G); V;
Vertex-set of G
> Degree(V.4);
1
> V1:=Vertices(G); V1;
{@ 1, 2, 3, 4, 5, 6 @}
> Degree(V1[4]);
1
> Type(V); Type(V1);
GrphVertSet
SetIndx
> Type(1); Type(V!1);
RngIntElt
GrphVert
> v:=V!1; Degree(v);
```

```
3
```

If we have created V as a vertex set, then we access the ith element of V using `V.i` or `V!i`. However, if we have created V as an indexed set via `Vertices`, we can access the ith element of V using `V[i]`.

There is an important point here: `Vertices` and `VertexSet` return a collection of vertices, even though they look like integers. To get the integer corresponding to a vertex, use the command `Index`.

```
> G:=GewirtzGraph();
> #Vertices(G);
56
> V:=Vertices(G);
> W:=VertexSet(G);
> W.1;
1
> Type(W.1);
GrphVert
> Type(V[1]);
GrphVert
> V[1];
1
> V[1]+V[2];

>> V[1]+V[2];
       ^

Runtime error in '+': Bad argument types
Argument types given: GrphVert, GrphVert

> Index(V[1])+Index(V[2]);
3
```

Edges are similarly dealt with. To access the edges, use `EdgeSet`, or `Edges`. Each element of this is an edge, and you have to use `EndVertices` to get the set of vertices. These are still vertices, not integers, even though one may ask if 1 is in the set, as below. (This is another example of Magma automatically altering which universe your variable belongs to if it can.) However, this does not work for the edge itself.

```
> E:=EdgeSet(G);
> E.1;
{1, 2}
> Type(E);
```

```
GrphEdgeSet
> E.1;
{1, 2}
> A:=EndVertices(E.1);
> A;
{ 1, 2 }
> Type(E.1);
GrphEdge
> Type(A);
SetEnum
> 1 in A;
true
> 1 in E.1;

>> 1 in E.1;
     ^

Runtime error in 'in': Bad argument types
Argument types given: RngIntElt, GrphEdge

> V!1 in E.1;
true
```

As we saw above, you can access the degree of a vertex v with `Degree(v)`.

```
> G:=Graph<6|{1,2},{1,3},{1,4},{2,3},{2,5},{3,6},{5,6}>;
> V:=VertexSet(G);
> MaximumDegree(G);
3 1
> MinimumDegree(G);
1 4
> DegreeSequence(G);
[ 0, 1, 2, 3 ]
> Neighbours(V.2);
{ 1, 3, 5 }
> IncidentEdges(V.6);
{ {6, 5}, {6, 3} }
```

There are similar functions for digraphs.

We can also examine the connectedness of a graph. One may test a graph's connectedness with `IsConnected(G)`, and find its connected components with `Components(G)`. The set of cut vertices is returned by `CutVertices(G)`, and a minimal set of vertices that disconnects the graph is returned by `VertexSeparator(G)`. The two commands `VertexConnectivity(G)` and `IsKVertexConnected(G,k)` return the connectivity of G and whether G is k-connected or not. All of the same commands exist with 'vertex' replaced by 'edge'.

```
> G:=Graph<6|{1,2},{1,3},{1,4},{2,3},{2,4},{2,5},{3,6},{5,6}>;
> VertexSeparator(G);
[ 5, 3 ]
> EdgeConnectivity(G);
2
```

There are many other commands for dealing with graphs, and the reader should have a look at the Magma on-line help to get a flavour for what sort of functions exist.

Exercise 3.6 How many isomorphism classes of graph are there on four points? On seven points? (`IsIsomorphic` does what you expect it to do.)

Exercise 3.7 Assume that each edge of a random graph G exists with probability $p = 0.6$. Derive an estimate for the probability that such a random graph on five points is connected. (A thousand tests should be sufficient.)

3.4 Generating Functions

Generating functions are important in combinatorics since they can be used to encode various sequences of numbers, such as the partition function. To accomplish this, we need to look at power series rings. This will serve as a useful introduction to other types of rings in the next chapter, such as polynomial rings.

The power series ring in one variable is easily constructed. As well as the ring over which power series are taken, there is an optional variable for the precision, set by default to 20.

```
> Z:=Integers();
> R<x>:=PowerSeriesRing(Z,10);
> R!(1/(1+x));
1 - x + x^2 - x^3 + x^4 - x^5 + x^6 - x^7 + x^8 - x^9 + O(x^10)
```

Magma can do arithmetic using power series, and unlike floating point techniques, the resulting errors do not magnify, due to the way in which power series arithmetic works. All of the commands available are in the relevant section in the Magma help file (under the section 'Local Arithmetic Fields'), but here are a few to demonstrate its use. (It should also be noted that the ever-useful hypergeometric series are implemented.)

```
> Q:=Rationals();
> R<x>:=PowerSeriesRing(Q,15);
> Sin(x);
x - 1/6*x^3 + 1/120*x^5 - 1/5040*x^7 + 1/362880*x^9 - 1/39916800*x^11 +
    1/6227020800*x^13 + O(x^15)
> Exp(x^2);
1 + x^2 + 1/2*x^4 + 1/6*x^6 + 1/24*x^8 + 1/120*x^10 + 1/720*x^12 + 1/5040*x^14 +
    O(x^15)
> Log(1+x);
x - 1/2*x^2 + 1/3*x^3 - 1/4*x^4 + 1/5*x^5 - 1/6*x^6 + 1/7*x^7 - 1/8*x^8 + 1/9*x^9
    - 1/10*x^10 + 1/11*x^11 - 1/12*x^12 + 1/13*x^13 - 1/14*x^14 + O(x^15)
> Laplace(Sin(x));
x - x^3 + x^5 - x^7 + x^9 - x^11 + x^13 + O(x^15)
```

[The command `Laplace` takes the Laplace transform of the power series given, and returns it as a power series itself.]

Power series rings are the first of three types of power series ring in Magma, which are stored internally as the same thing, namely a Puiseux series ring: we will define this momentarily. The second type of series ring is a Laurent series ring, and the syntax is similar to the above.

```
> Q:=Rationals();
> R<z>:=LaurentSeriesRing(Q,15);
> Sin(z)/z^5;
z^-4 - 1/6*z^-2 + 1/120 - 1/5040*z^2 + 1/362880*z^4 - 1/39916800*z^6 +
    1/6227020800*z^8 - 1/1307674368000*z^10 + O(z^11)
> Derivative(Sin(z)/z^5);
-4*z^-5 + 1/3*z^-3 - 1/2520*z + 1/90720*z^3 - 1/6652800*z^5 + 1/778377600*z^7 -
    1/130767436800*z^9 + O(z^10)
```

Notice how the precision shifts in this case: for $\sin z/z^5$, the point at which precision is lost is z^{11}, whereas when you take the derivative, it drops to z^{10}. This is because each element of the ring can have fifteen coefficients, and the lowest one moved down by one, so the maximum power had to decrease.

A Puiseux series ring can have fractional powers. These are built into Magma, although we will not need them in this course.

The reason why power series rings are considered here is that many sequences in combinatorics can be defined using generating functions. The power series rings can be used to find the coefficients of the generating functions, at least up until a certain predetermined point. The final exercises of this chapter contain such a generating function.

Exercise 3.8 The generating function for the Schröder numbers is

$$G(x) = \frac{1 - x - (1 - 6x + x^2)^{1/2}}{2x}.$$

What are the first thirty Schröder numbers?

Extended Exercise 3.9 In this exercise, we will prove Euler's result again as in Exercise 3.3, but without resorting to sorting through the partitions generated by `Partitions(n)`. Construct both functions from scratch, and use them to confirm Euler's result up to $n = 60$. (This will be difficult using your previous functions!) To see how long your program takes to run, use the `time` command in front of the function. For example:

```
time for i in [1..60] do
for> #SelfConjugatePartitions(i) eq #OddDistinctPartitions(n);
for> end for;
```

Chapter 4

Number Theory and Linear Algebra

We begin by looking at number theory, and firstly examining fields. Magma has implementations of rationals and reals, as we have seen, and also of finite fields. It has a massive database of functions concerning Galois theory and class field theory. Since number theory is not our main concern, we will not delve deeply into this subject.

4.1 Real and Complex Numbers

In the very first chapter we introduced the integers, rationals, reals and complexes. We will now look at some of the more complicated commands available for these rings. We begin with the integers. There are quite a few commands for testing properties of integers.

```
> x:=5; y:=15; z:=4;
> IsOdd(x); IsEven(x);
true
false
> IsSquare(z);IsSquarefree(y);
true 2
true
> IsPrime(x),IsPrime(y);
true false
```

The function `IsPrime` is not probabilistic, and so returns true if and only if the integer is prime.

Taking integral logarithms is easy in Magma, as is taking normal logarithms.

```
> Ilog2(17);
4
> Ilog(3,82);
4
> Log(17);
```

```
2.83321334405621608024953461787
```

Magma can produce random integers in an interval. The command `Random(n)` produces a random number between 0 and n inclusive, and with two arguments a and b, it chooses integers from the closed interval $[a, b]$.

```
> Random(5);
0
> Random(5);
4
> Random(5,15);
11
> X:=[Random(1,6):i in [1..100]];
> &+X/100.0;
3.44000000000000000000000000000
```

Computing the factorization of an integer is very easy (for small integers!) in Magma. However, computers' definition of small may differ from humans'. Magma also possesses numerous factorization algorithms, which we will not concern ourselves with here. Related to factorization is the collection of all possible divisors of a number, which can easily be computed in Magma. In turn, the Möbius function is related to this.

```
> time Factorization(Random(10^50,2*10^50));
[ <356351, 1>, <18399288162473693, 1>, <17362560661165429873826250773, 1> ]
Time: 0.210
> Divisors(144);
[ 1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 36, 48, 72, 144 ]
> MoebiusMu(6);
1
> MoebiusMu(30);
-1
```

Finally for the integers, we consider modular arithmetic. We have already seen the `mod` function, but there are other, more specific functions. In some situations, it is useful, given n, k and m, to take the number n^k modulo m. This is given by `Modexp(n,k,m)`. While this function would be easy to replicate oneself, Magma also has the ability to invert those elements that may be inverted in the ring $\mathbb{Z}/m\mathbb{Z}$. It square roots exist, then Magma can retrieve them. (It returns an error if they do not.) It can also tell you the order of an invertible element in the group of units of $\mathbb{Z}/m\mathbb{Z}$, and hence whether it is a primitive element or not.

```
> Modinv(5,14);
```

```
3
> Modsqrt(-1,13);
8
> Modsqrt(-1,11);

>> Modsqrt(-1,11);
          ^

Runtime error in 'Modsqrt': Argument has no square root

> Modorder(5,9);
6
> IsPrimitive(5,9);
true
> Modorder(4,9);
3
> IsPrimitive(4,9);
false
```

Magma has implemented the Chinese Remainder Theorem, which may be accessed via `CRT`. It can also solve a general linear congruence of the form $ax \equiv b \mod n$ using the command `Solution(a,b,m)`. This returns two values, y and k, such that all solutions of the congruence are of the form $y + ik$ for i an integer.

```
> Solution(4,12,24);
3 6
> for i in [0..3] do ((3+i*6)*4) mod 24; end for;
12
12
12
12
> CRT([3,4],[7,9]);
31
> 31 mod 7,31 mod 9;
3 4
```

The real and complex numbers are more difficult in Magma, if only because these can only be dealt with to a certain level of precision. We saw this right back in the first chapter. Here we will examine this more closely.

The first observation to make is that Magma lies!

```
> R:=RealField(40);
> R!0.098765432109876543210987654321098765432;
0.09876543210987654321098765432105998087752
```

The number input into Magma has 39 decimal digits, so should not have been rounded off, but that's not what came out at the end. In fact, what came out at the end was only accurate to thirty digits. This is because the number on the right-hand side was originally constructed inside Magma with thirty digits of precision in Magma's *default field*, and then sent over to our field R with more than thirty digits of precision.

```
> R:=RealField(40);
> GetDefaultRealField();
Real field of precision 30
> SetDefaultRealField(RealField(100));
> R!0.098765432109876543210987654321098765432;
0.09876543210987654321098765432109876543201
```

Even after changing the default field, Magma is still producing a little error at the end. It's considerably less than before, but not nice. This little error is because the 40 digits of precision specified in the definition of R are translated into a condition on *binary* precision, which is the way that Magma stores reals internally. (Essentially, Magma stores a binary expansion of the real number to a certain point.)

There are many functions built directly into Magma. It also possesses a few constants. Here we include a few such functions. For a complete list, see the Magma help files.

```
> R:=RealField(40);
> Pi(R);
3.141592653589793238462643383279502884197
> EulerGamma(R);
0.5772156649015328606065120900824024310421
> Catalan(R);
0.9159655941772190150546035149323841107742
> Gamma(0.5);
1.77245385090551602729816748334
> Sin(3.14);
0.00159265291648695254054143632495
> Erf(4);
0.999999984582742099719981147840
```

If you want to construct a continued fraction corresponding to a real number, Magma can perform this for you. It depends somewhat on the default field.

```
> ContinuedFraction(Pi(R));
[ 3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2, 2, 2, 1, 84, 2, 1,
1, 15, 3, 13, 1, 4, 2, 6, 6, 100, 24, 1, 3, 1, 1, 1, 6, 1, 12, 4, 1, 15, 47 ]
> 3+1/(7+1/(15+1/292))*1.0;
3.14150973868665008559708000905
> ContinuedFraction(Sqrt(2));
[ 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 5, 1, 1, 15, 7, 1, 1, 2 ]
> SetDefaultRealField(RealField(100));
> ContinuedFraction(Sqrt(2));
[ 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 10,
225, 1, 2, 156, 4 ]
```

Finally for this section, we show how to move between Cartesian and polar co-ordinates, and also how to extract a little information from a complex number.

```
> C<i>:=ComplexField(30);
> ComplexToPolar(3+3*i);
4.24264068711928514640506617263 0.785398163397448309615660845819
> PolarToComplex(1,Pi(R)/3);
0.500000000000000000000000000000000000000 +
0.866025403784438646763723170752936183471 4*$.1
> x:=3+3*i;
> Modulus(x);
4.24264068711928514640506617263
> Arg(x);
0.785398163397448309615660845820
> Re(x),Im(x);
3.00000000000000000000000000000 3.00000000000000000000000000000
```

Real functions such as $\sin x$ that are extended to the complex plane can, in Magma, be evaluated on any complex number usually in their domain.

```
> Sin(4+i);
-1.16780727488951847719094617152 - 0.768162763456573127766833209176*i
> Log(i);
1.57079632679489661923132169164*i
```

```
> Log(-1);

>> Log(-1);
      ^
Runtime error in 'Log': Argument 1 is not positive

> Log(C!-1);
3.14159265358979323846264338328*i
```

4.2 Polynomials

In the final section of the previous chapter, we saw the power series ring in one variable being constructed, as well as two other associated series rings. In this short section, we will discuss polynomial rings and their quotients.

Magma can deal with polynomial rings quite effectively. However, there is a subtle difference between a univariate polynomial ring and a multivariate polynomial ring that happens to only have one indeterminate.

```
> R<x>:=PolynomialRing(Rationals());
> x^2+1 in R;
true
> (x^2+1)*(x^6+1);
x^8 + x^6 + x^2 + 1
> (x^8+x^6+x^2+1)/(x^2+1);
x^6 + 1
```

Accessing the terms and coefficients of a polynomial is a difficult business in Magma, not because it is not possible, but that there is a bewildering array of functions that you can use. To get a specific coefficient, or the leading coefficient or term, you can use the code as given in the example below.

```
> Coefficient(2*x^3+1/2,0);
1/2
> LeadingCoefficient(3*x^3+1);
3
> LeadingTerm(3*x^3+1);
3*x^3
```

The same is possible with the trailing term.

One may also turn a polynomial into a sequence, or coefficients, terms or monomials.

```
> Terms(2*x^3+1/2);
[
    1/2,
    2*x^3
]
> Coefficients(2*x^3+1/2);
[ 1/2, 0, 0, 2 ]
> Monomials(2*x^3+1/2);
[
    1,
    x,
    x^2,
    x^3
]
```

This should be enough for you to do whatever you need to do with the coefficients of a polynomial.

It is possible to factorize a polynomial in Magma. The linear factors correspond to roots, although they can be produced using a dedicated command.

```
> Factorization(x^3-1);
[
    <x - 1, 1>,
    <x^2 + x + 1, 1>
]
> Roots(x^2-1);
[ <-1, 1>, <1, 1> ]
> Roots((x^2-1)^2);
[ <-1, 2>, <1, 2> ]
> IsIrreducible(x^3-2*x+1/3);
true
> Discriminant(x^3-2*x+1/3);
29
> CompanionMatrix(x^3-2*x+1/3);
[   0    1    0]
[   0    0    1]
[-1/3    2    0]
```

Elementary calculus can be performed on univariate polynomial rings.

```
> Derivative(x^4+3*x^3-2*x);
4*x^3 + 9*x^2 - 2
```

```
> Integral(x^4+3*x^3-2*x);
1/5*x^5 + 3/4*x^4 - x^2
```

Much of this makes sense for multivariate polynomial rings. To define them is easy.

```
> R<x,y>:=PolynomialRing(Rationals(),2);
> x*y+1;
x*y + 1
> Factorization(x^2*y^3 + x^2*y^2 + x^2*y + x^2);
[
    <y + 1, 1>,
    <y^2 + 1, 1>,
    <x, 2>
]
```

However, a univariate polynomial ring is different from a multivariate polynomial ring in one variable.

```
> R<x>:=PolynomialRing(Rationals());
> S<y>:=PolynomialRing(Rationals(),1);
> Derivative(x);
1
> Derivative(y);

>> Derivative(y);
             ^
Runtime error in 'Derivative': Bad argument types
Argument types given: RngMPolElt
```

Partial differentiation and integration is performed by including the variable with respect to which you intend to perform calculus.

We now deal with ideals and quotients of rings. Defining an ideal is simply a matter of giving its generators.

```
> I:=ideal<R|f>;
> x in I;
false
> J:=ideal<R|g>;
> I meet J;
Ideal of Univariate Polynomial Ring in x over Rational Field generated by x^8 +
    x^5 + x^4 + x^3 + x + 1
> R eq I+J;
true
> I*J subset (I meet J);
true
```

It is a similar deal with quotients. Here we may give the generators of the quotient ring, and they will match up with the images of the generators of the original ring. When doing arithmetic in the quotient ring, Magma automatically reduces expressions via the ideal.

```
> R<x,y,z>:=PolynomialRing(Integers(),3);
> S<a,b,c>:=quo<R|x*y*z,x^2+y^2>;
> (a*b+b*c)*c;
b*c^2
```

This ability of Magma is extremely powerful, and makes working in quotients of polynomial rings much easier.

Exercise 4.1 Magma (for some reason) does not have a function to create a Laurent polynomial ring (either univariate or multivariate). Create a function which performs this task as well as you can, using the tools available in this section.

4.3 Fields

To construct a finite field of order q, one simply types in `GF(q)`. To specify a generator for the multiplicative group, we again do what we did in the previous section.

```
> F<x>:=GF(4);
> for i in F do i; end for;
1
x
x^2
0
```

Arithmetic in finite fields is done by Zech logarithms for small finite fields, and using a polynomial ring for large fields. For the prime 2, for example, all fields of order at most 2^20 use tables of Zech logarithms to do field arithmetic, and consequently field operations can be done very quickly. We will demonstrate this difference in speed by timing some field operations.

```
> for i in [18..22] do
for> F<x>:=GF(2^i);
for> time for j in F do if(j ne 0) then y:=j^-1; end if; end for;
for> end for;
Time: 0.160
Time: 0.320
Time: 0.670
Time: 7.500
Time: 15.060
```

The irreducible polynomials that are used in defining finite fields are so-called *Conway polynomials*, which enjoy special consistency properties with smaller Conway polynomials. Tables of Conway polynomials are known for degree at most 92 for $p = 2$, for degree at most 44 for $p = 3$, and so on. If one is interested, the polynomials may be retrieved with `ConwayPolynomial(p,n)`.

To calculate the minimal polynomial of a particular element of a field, you simply use the command `MinimalPolynomial`.

```
> F:=GF(8);
> F<x>:=GF(8);
> MinimalPolynomial(Random(F));
$.1^3 + $.1 + 1
> MinimalPolynomial(Random(F));
$.1^3 + $.1 + 1
> MinimalPolynomial(Random(F));
$.1 + 1
```

Constructing splitting fields of polynomials is very easy.

```
> F:=GF(2);
> R<x>:=PolynomialRing(F);
> SplittingField(x^5+x^2+x+1);
Finite field of size 2^3
> R<x>:=PolynomialRing(Rationals());
> F:=SplittingField(x^2+1);
> F;
Number Field with defining polynomial x^2 + 1 over the Rational Field
> F<i>:=SplittingField(x^2+1);
> i;
i
> i^2;
-1
```

Cyclotomic fields are implemented in Magma as well. To obtain the field got by adjoining a primitive nth root of unity, simply use the command `CyclotomicField(n)`.

```
> F:=CyclotomicField(6);
> R<x>:=PolynomialRing(F);
> Factorization(x^6-1);
[
    <x - 1, 1>,
    <x + 1, 1>,
    <x - zeta_6, 1>,
    <x - zeta_6 + 1, 1>,
    <x + zeta_6 - 1, 1>,
    <x + zeta_6, 1>
]
> K:=CyclotomicField(7);
> Degree(K);
6
```

To find the nth cyclotomic polynomial is also very easy.

```
> CyclotomicPolynomial(6);
x^2 - x + 1
> CyclotomicPolynomial(7);
x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
```

Exercise 4.2 How many irreducible polynomials are there of degree at most 5 over $\mathbb{F}_4$?

4.4 Galois Theory

To find the Galois group of a square-free polynomial is quite easy in Magma.

```
> R<x>:=PolynomialRing(Integers());
> GaloisGroup(x^2+1);
Symmetric group acting on a set of cardinality 2
Order = 2
    (1, 2)
[ $.1 + 1 + O(3^5), -$.1 - 1 + O(3^5) ]
GaloisData over Z_3

> GaloisGroup(x^5+x^3+x+1);
Symmetric group acting on a set of cardinality 5
Order = 120 = 2^3 * 3 * 5
[ -158153*$.1^5 - 134964*$.1^4 + 39442*$.1^3 - 47438*$.1^2 - 153279*$.1 +
    161570 + O(13^5), 126954*$.1^5 + 160315*$.1^4 + 87195*$.1^3 -
    18248*$.1^2 - 118402*$.1 - 80789 + O(13^5), 31199*$.1^5 - 25351*$.1^4
    - 126637*$.1^3 + 65686*$.1^2 - 99612*$.1 - 109499 + O(13^5),
    -20263*$.1^5 + 78350*$.1^4 - 137002*$.1^3 + 159419*$.1^2 - 141996*$.1
    - 96182 + O(13^5), 20263*$.1^5 - 78350*$.1^4 + 137002*$.1^3 -
    159419*$.1^2 + 141996*$.1 + 124900 + O(13^5) ]
GaloisData over Z_13

> GaloisGroup(x^5-5*x+4);

>> GaloisGroup(x^5-5*x+4);
              ^
Runtime error in 'GaloisGroup': Polynomial must be squarefree
```

Given some Galois extension of $\mathbb{Q}$, Magma can calculate its Galois group as well.

```
> Q:=Rationals();
> R<x>:=PolynomialRing(Q);
> f:=x^4+1;
> K:=SplittingField(f);
> GaloisGroup(K);
Permutation group acting on a set of cardinality 4
Order = 4 = 2^2
    (1, 3)(2, 4)
    (1, 4)(2, 3)
[ -608190 + O(17^5), -146991 + O(17^5), 146991 + O(17^5), 608190 + O(17^5) ]
GaloisData over Z_17
```

The other two pieces of data returned are a by-product of the way that Magma calculates Galois groups; it goes via a p-adic field (for some suitable prime p) and these outputs are not of interest for us.

It is also possible to compute the ring of integers $\mathcal{O}$ of an algebraic number field K, and compute invariants of it, like its class number.

```
> Q:=Rationals();
> R<x>:=PolynomialRing(Q);
> f:=x^4-1;
> K:=SplittingField(f);
> R:=RingOfIntegers(K);
> ClassNumber(R);
1
> f:=x^23-1;
> K:=SplittingField(f);
> R:=RingOfIntegers(K);
> time ClassNumber(G);
3
Time: 6034.560
```

(It is not recommended that you try this example out yourself, since the computational time is considerable.)

Exercise 4.3 Let f be an irreducible polynomial with integer coefficients. It is well-known that if f is irreducible over $\mathbb{Q}$, then the Galois group of f over $\mathbb{Q}$ is a transitive subgroup of degree n, where n is the degree of the polynomial. Above we gave an example of a polynomial with Galois group S_5. Find quintic polynomials with Galois groups the other transitive subgroups of S_5, namely:

(i) A_5;

(ii) $C_5 \rtimes C_4$, the affine general linear group $\mathrm{AGL}_1(5)$;

(iii) D_{10}; and

(iv) (More difficult) C_5.

Exercise 4.4 Which of the rings of integers of quadratic fields $K = \mathbb{Q}(\sqrt{d})$ (d positive) between 2 and 20 are UFDs? (Remember that d must be square-free.) Which is the smallest d such that $\mathcal{O}_K$ has class number 3? How about 4?

Exercise 4.5 There are exactly four integer solutions to the equation $y^3 = x^2 + 2000000$. Find them.

Extended Exercise 4.6 Here we will determine which integers between 1 and 30 are congruent numbers (i.e., the area of a right-angled triangle with rational sides) and find some of the corresponding triangles.

It is a fact that an integer n is congruent if and only if the elliptic curve

$$y^2 = x^3 - n^2x = x(x+n)(x-n)$$

has infinitely many rational points. (It clearly has the three points $(x, y) = (0, 0)$, $(-n, 0)$ and $(n, 0)$, together with the point at infinity guaranteeing that the Mordell–Weil group of the elliptic curve has the Klein four group as a subgroup.)

(i) The function `EllipticCurve([a,b])` constructs the elliptic curve

$$y^2 = x^3 + ax + b.$$

The rank of the Mordell–Weil group of an elliptic curve E is given by `MordellWeilRank(E)`. Determine which of the integers between 1 and 30 are congruent.

(ii) If you have performed the first part correctly, you will have found that 7 is a congruent number. The rational points on an elliptic curve can be obtained using the command

```
RationalPoints(E:Bound:=i)
```

where `i` is some integer that bounds the size of the points. Doubling a non-trivial point on the elliptic point gives a way of finding a triangle that confirms that the number is congruent. Let X be the x-co-ordinate of a point on the elliptic curve (with X a square). Define

$$X = a^2, \quad X - n = b^2, \quad X + n = c^2, \quad \alpha = c + b, \quad \beta = c - b, \quad \gamma = 2a.$$

Then $\alpha\beta = 2n$, and $\alpha^2 + \beta^2 = \gamma^2$, as required.

Find α, β and γ for $n = 7$. If you are interested, find them for other values of n.

4.5 Mappings and Homomorphisms

Mappings are a fundamental part of Magma, and the reason that we haven't seen them until now is simply chance. There are three types of mappings in use in Magma: maps, homomorphisms, and partial maps. While we will not meet partial maps here, we will examine the other two.

A map is simply a function from a domain to a codomain. To specify a map, you need to give a collection of pairs (a, b) where $a \in A$, the domain, and $b \in B$, the codomain. Every element of A must appear exactly once in such a collection. We have not seen tuples before, but they are not difficult to define.

```
> A:=[1,2,3];
> B:=[4,5,6];
> X:=CartesianProduct(A,B);
> Random(X);
<3, 5>
> map<A -> B | [<1,4>,<2,4>,<3,5>]>;
Mapping from: SeqEnum: A to SeqEnum: B
    <1, 4>
    <2, 4>
    <3, 5>
```

Maps may also be defined by a general rule, as we demonstrate shortly. We include a strange fact about maps: namely, that two maps that are equal do not register as being equal.

```
> g:=map<A -> B |x:->x+3>;
> h:=map<A -> B | [<1,4>,<2,5>,<3,6>]>;
> g eq h;
```

```
false
> for i in A do g(i) eq h(i); end for;
true
true
true
> f:=map<A -> B | [<1,4>,<2,5>,<3,6>]>;
> f eq h;
false
```

The images of elements can be given either in the standard mathematical way, or using the 'commercial at' symbol.

```
> f:=map<A -> B | [<1,4>,<2,4>,<3,5>]>;
> f(1);
4
> 3@f;
5
```

Given a map `f`, the functions `Domain(f)`, `Codomain(f)`, `Kernel(f)`, and `Image(f)` all do the expected thing. They tend not to work for infinite objects, though.

```
> Z:=Integers();
> f:=map<Z->Z|x:-> 3*x>;
> Kernel(f);

>> Kernel(f);
         ^
Runtime error in 'Kernel': Kernel is not computable or representable
```

As mentioned before, the other type of mapping is a homomorphism. Typically these are defined only on the generators, and typically it is up to the user to ensure that the mapping specified really *is* a homomorphism. Precisely how homomorphisms are defined depends upon the algebraic structure on which you are trying to define them, but a typical example is that of a group. In that case, a homomorphism is defined, for example, by a list of tuples `<g,h>`, where `g` lies in the domain and `h` in the codomain.

4.6 Matrices

Constructing matrices is not a difficult task in Magma. The main problem is to decide which of the many different methods to use! Here we demonstrate the main types by example, and leave it to the reader to grasp the obvious syntax.

```
> Z:=Integers();
> Q:=Rationals();
> R:=RealField(20);
> Matrix(Z,3,2,[1,2,3,4,5,6]);
[1 2]
[3 4]
[5 6]
> Matrix(R,3,2,[1,2,3,4,5,6]);
[1.0000000000000000000 2.0000000000000000000]
[3.0000000000000000000 4.0000000000000000000]
[5.0000000000000000000 6.0000000000000000000]
> Matrix(3,2,[1,2,3,4,5,6]);
[1 2]
[3 4]
[5 6]
> Matrix([[1,2],[3,4],[5,6]]);
[1 2]
[3 4]
[5 6]
> Matrix(R,[[1,2],[3,4],[5,6]]);
[1.0000000000000000000 2.0000000000000000000]
[3.0000000000000000000 4.0000000000000000000]
[5.0000000000000000000 6.0000000000000000000]
> Matrix(Z,[[1,4/2],[3,4],[5,6]]);
[1 2]
[3 4]
[5 6]
> Matrix(3,[1,2,3,4,5,6]);
[1 2 3]
[4 5 6]
> Matrix(4,[1,2,3,4,5,6,7,8,9,10,11,12]);
[ 1  2  3  4]
[ 5  6  7  8]
[ 9 10 11 12]
> DiagonalMatrix([1,2,3,4]);
[1 0 0 0]
[0 2 0 0]
[0 0 3 0]
```

```
[0 0 0 4]
> UpperTriangularMatrix([1,2,3,4,5,6]);
[1 2 3]
[0 4 5]
[0 0 6]
> SymmetricMatrix(GF(11),[1,0,-1,4,6,8]);
[ 1  0  4]
[ 0 10  6]
[ 4  6  8]
> AntisymmetricMatrix(GF(11),[1,0,-1,4,6,8]);
[ 0 10  0  7]
[ 1  0  1  5]
[ 0 10  0  3]
[ 4  6  8  0]
> PermutationMatrix(Z,[3,2,4,1,6,5]);
[0 0 1 0 0 0]
[0 1 0 0 0 0]
[0 0 0 1 0 0]
[1 0 0 0 0 0]
[0 0 0 0 0 1]
[0 0 0 0 1 0]
```

Constructing a vector is also easy.

```
> Vector([3,6,2,9]);
(3 6 2 9)
> Vector(GF(3),[3,6,2,9]);
(0 0 2 0)
```

It is possible to alter the entries in a matrix, both individually and row-by-row. If changing a whole row, a vector is needed.

```
> X:=AntisymmetricMatrix(GF(11),[1,0,-1,4,6,8]);
> X;
[ 0 10  0  7]
[ 1  0  1  5]
[ 0 10  0  3]
[ 4  6  8  0]
> X[2];
( 1  0  1  5)
> X[2,1];
```

```
1
> X[2,3]:=3;
> X;
[ 0 10  0  7]
[ 1  0  3  5]
[ 0 10  0  3]
[ 4  6  8  0]
> X[3]:=Vector(GF(11),[3,4,1,2]);
> X;
[ 0 10  0  7]
[ 1  0  3  5]
[ 3  4  1  2]
[ 4  6  8  0]
```

Multiplying and adding matrices is performed using `*` and `+`, and the inverse and transpose of a matrix `A` are given by `A^-1` and `Transpose(A)` respectively. The determinant and trace of a matrix can be found using the obvious commands.

There are good tools for performing Gaussian elimination, in the sense that commands for adding one row to another, swapping two rows, and scaling a row are available. To add c copies of row i to row j in the matrix A, the command `AddRow(A,c,i,j)`. The other commands have obvious syntax from the example below.

```
> X:=Matrix(4,[0,-1,0,-4,1,0,3,5,3,4,1,2,4,6,8,0]);
> X;
[ 0 -1  0 -4]
[ 1  0  3  5]
[ 3  4  1  2]
[ 4  6  8  0]
> AddRow(X,4,1,2);
[  0  -1   0  -4]
[  1  -4   3 -11]
[  3   4   1   2]
[  4   6   8   0]
> SwapRows(~X,1,2); X;
[ 1  0  3  5]
[ 0 -1  0 -4]
[ 3  4  1  2]
[ 4  6  8  0]
> MultiplyRow(X,3,4);
[ 1  0  3  5]
[ 0 -1  0 -4]
[ 3  4  1  2]
[12 18 24  0]
```

All of these functions are also available for the columns of the matrix.

Finding the Jordan normal form for a matrix is very easy, since it is a simple command.

```
> X:=Matrix(GF(11),4,[0,-1,0,-4,1,0,3,5,3,4,1,2,4,6,8,0]);
> X;
[ 0 10  0  7]
[ 1  0  3  5]
[ 3  4  1  2]
[ 4  6  8  0]
> JordanForm(X);
[10  0  0  0]
[ 0  8  1  0]
[ 0  0  8  1]
[ 0  0  0  8]
> RationalForm(X);
[ 0  1  0  0]
[ 0  0  1  0]
[ 0  0  0  1]
[ 6  1  8  1]
```

Magma can also produce the Smith normal form, together with the two unimodular matrices that multiply either side to achieve the Smith form.

```
> X:=Matrix(4,[0,-1,0,-4,1,0,3,5,3,4,1,2,4,6,8,0]);
> S,P,Q:=SmithForm(X);
> P*X*Q;
[  1   0   0   0]
[  0   1   0   0]
[  0   0   1   0]
[  0   0   0 236]
> S;
[  1   0   0   0]
[  0   1   0   0]
[  0   0   1   0]
[  0   0   0 236]
```

Finally, constructing the eigenvalues and eigenvectors of a matrix is possible. Magma will only construct the eigenvalues that lie in the ring over which you have defined your matrix.

```
> X:=Matrix(GF(11),4,[0,-1,0,-4,1,0,3,5,3,4,1,2,4,6,8,0]); X;
[ 0 10  0  7]
[ 1  0  3  5]
[ 3  4  1  2]
[ 4  6  8  0]
> Eigenvalues(X);
{ <8, 3>, <10, 1> }
> R<x>:=PolynomialRing(GF(11));
> CharacteristicPolynomial(X);
x^4 + 10*x^3 + 3*x^2 + 10*x + 5
> Y:=Matrix(4,[0,-1,0,-4,1,0,3,5,3,4,1,2,4,6,8,0]);
> Eigenvalues(Y);
{}
> Y:=Matrix(R,4,4,[0,-1,0,-4,1,0,3,5,3,4,1,2,4,6,8,0]);
> Eigenvalues(Y);
{ <6.9215307718573629707, 1>, <2.0668501435851666781, 1> }
```

The eigenvectors are calculable via the eigenspaces of a matrix.

```
> X:=Matrix(GF(11),4,[0,-1,0,-4,1,0,3,5,3,4,1,2,4,6,8,0]);
> Eigenvalues(X);
{ <8, 3>, <10, 1> }
> Eigenspace(X,8);
Vector space of degree 4, dimension 1 over GF(11)
Echelonized basis:
( 1  1  3  5)
> Eigenspace(X,-1);
Vector space of degree 4, dimension 1 over GF(11)
Echelonized basis:
( 1  3  3  5)
> JordanForm(X);
[10  0  0  0]
[ 0  8  1  0]
[ 0  0  8  1]
[ 0  0  0  8]
> v:=Vector(GF(11),[1,1,3,5]);
> 8*v;
( 8  8  2  7)
> v*X;
( 8  8  2  7)
```

There are also commands available for calculating minors, for deleting rows and columns, for building block matrices, and other things.

Exercise 4.7 Find the determinant of the matrix

$$\begin{pmatrix} x & x+x^{-1} & y+xy+y^{-1} \\ 1 & xy^2-yx^2 & xy^{-1} \\ x^2+x^{-2} & 1/2 & (xy)^{-1}-y^2 \end{pmatrix}$$

in the Laurent polynomial ring in x and y over $\mathbb{Q}$.

Extended Exercise 4.8 Write a program that will perform Gaussian elimination on a matrix X, and code your program so that it outputs both the original matrix in row echelon form and the inverse of the matrix, should it exist. (The functions `NumberOfRows`, `NumberOfColumns`, and `Rank` might be useful here.) Check that your program does indeed produce the inverse with a few examples.

4.7 Vector Spaces

Vector spaces are relatively easy to define in Magma, although there aren't all that many interesting things you can do with them. Vector spaces and individual elements of vector spaces are easy to define.

```
> K:=Rationals();
> V:=VectorSpace(K,4);
> elt<V|5,1/2,0,-3>;
(  5 1/2   0  -3)
> V![5,1/2,0,-3];
(  5 1/2   0  -3)
```

To get a random element of a vector space, the space ought be finite.

```
> K<w>:=GF(9);
> V:=VectorSpace(K,6);
> Random(V);
(w^3   w   2 w^7   2   1)
> F:=Rationals();
> W:=VectorSpace(F,6);
> Random(W);


>> Random(W);
         ^
```

```
Runtime error in 'Random': Coefficient ring of argument 1 has no random
algorithm
```

Vectors may be added and subtracted, using + and -, and multiplied by a scalar.

```
> V:=VectorSpace(Rationals(),6);
> v:=V![1,0,-1,3,2,6];
> w:=V![7,7,5,-4,-3,-1/2];
> (v,w);           //The inner product
-19
> Norm(v);
51
> Normalize(w);
(    1     1   5/7  -4/7  -3/7 -1/14)
> TensorProduct(v,w);
(7 7 5 -4 -3 -1/2 0 0 0 0 0 0 -7 -7 -5 4 3 1/2 21 21 15 -12 -9 -3/2 14 14 10 -8
    -6 -1 42 42 30 -24 -18 -3)
> w[6];
-1/2
```

Defining subspaces is a simple task, assuming that you have a generating set.

```
> V:=VectorSpace(Rationals(),6);
> v:=V![1,0,-1,3,2,6];
> w:=V![7,7,5,-4,-3,-1/2];
> W:=sub<V|v,w>;
> W;
Vector space of degree 6, dimension 2 over Rational Field
Generators:
(   1    0   -1    3    2    6)
(   7    7    5   -4   -3 -1/2)
Echelonized basis:
(      1      0     -1      3      2      6)
(      0      1   12/7  -25/7  -17/7 -85/14)
> Basis(W);
[
    ( 1  0 -1  3  2  6),
    (      0      1   12/7  -25/7  -17/7 -85/14)
]
```

The construction of quotient spaces is equally easy.

```
> quo<V|v>;
Full Vector space of degree 5 over Rational Field
Mapping from: ModTupFld: V to Full Vector space of degree 5 over Rational Field
> U,phi:=quo<V|v>;
> u:=U![1,-2,1,0,1];
> u@@phi;
( 0  1 -2  1  0  1)
> v@phi;
(0 0 0 0 0)
> w@phi;
(    7    12   -25   -17 -85/2)
> w;
(   7    7    5   -4   -3 -1/2)
> v;
( 1  0 -1  3  2  6)
```

Vector space homomorphisms may be constructed as a set, and linear transformations construed as elements of that set.

```
> V:=VectorSpace(GF(5),4);
> W:=VectorSpace(GF(5),3);
> H:=Hom(V,W);
> phi:=Random(H);
> phi;
[4 1 4]
[2 0 1]
[4 2 1]
[2 3 1]
> v:=V![1,0,-1,3];
> v@phi;
(1 3 1)
> Kernel(phi);
Vector space of degree 4, dimension 1 over GF(5)
Echelonized basis:
(1 4 2 0)
> Image(phi);
Full Vector space of degree 3 over GF(5)
```

Chapter 5

Group Theory

In this chapter we will meet a lot of the techniques built into Magma for working with (predominantly finite) groups. Magma has very powerful tools for this, and hopefully by the end of this chapter some of these will be elucidated. It would be infeasible to include here even a small percentage of the functions available, but the goal is to give the basic techniques and make clear some of the fundamental concepts involved in the way that Magma handles groups.

5.1 Producing Groups

We begin by listing a few of the groups that are hard-coded in Magma. We include some basic commands for working with groups.

```
> G:=SymmetricGroup(5);
> Order(G);
120
> Random(G);
(1, 4, 3)
> H:=DihedralGroup(4);
> Generators(H);
{
    (1, 2, 3, 4),
    (1, 4)(2, 3)
}
> K:=CyclicGroup(7);
> L:=AlternatingGroup(4);
> Exponent(L);
6
> IsNilpotent(L);
false
```

```
> IsSoluble(L);
true
> IsSoluble(G);
false
```

Thus there are in-built commands for producing cyclic, dihedral, symmetric and alternating groups. The other commands used are self-explanatory. One may also construct an arbitrary permutation group. There are two ways of doing this, and we shall illustrate both.

```
> G:=SymmetricGroup(11);
> x:=G!(2,10)(4,11)(5,7)(8,9);
> y:=G!(1,4,3,8)(2,5,6,9);
> M:=sub<G|[x,y]>;
> N<a,b>:=PermutationGroup<11|(2,10)(4,11)(5,7)(8,9),(1,4,3,8)(2,5,6,9)>;
> a;
(2, 10)(4, 11)(5, 7)(8, 9)
> M eq N;
true
```

The `!` command was seen before as a way of coercing the object to the right to live inside the object to the left, if this is possible. On its own, a permutation doesn't mean anything.

```
> x:=(2,10)(4,11)(5,7)(8,9);


>> x:=(2,10)(4,11)(5,7)(8,9);
       ^

Runtime error in elt< ... >: No permutation group context in which to create cycle
```

We have also seen here how to define a subgroup of a group. We give another example, and introduce the notation for a quotient, which is not difficult to grasp.

```
> G<x,y>:=DihedralGroup(5);
> a;
(2, 10)(4, 11)(5, 7)(8, 9)
> x;
(1, 2, 3, 4, 5)
> y;
(1, 5)(2, 4)
> H:=sub<G|x>;
> H;
Permutation group H acting on a set of cardinality 5
    (1, 2, 3, 4, 5)
```

```
> G/H;
Permutation group acting on a set of cardinality 2
    Id($)
    (1, 2)
```

Attempting to quotient out by a subgroup that is not normal brings an error message.

```
> K:=sub<G|y>;
> G/K;

>> G/K;
    ^
Runtime error in '/': Subgroup is not a normal subgroup
```

In order to test whether a subgroup is normal, one uses the `IsNormal` command!

```
> IsNormal(G,K);
false
> IsNormal(G,H);
true
```

We have dealt with the creation of groups as permutation groups, and the definition of subgroups and quotients. Of course, there are other ways of producing new subgroups. Here we give the code for the intersection and join of two subgroups.

```
> G<x,y>:=DihedralGroup(4);
> x,y;
(1, 2, 3, 4)
(1, 4)(2, 3)
> H:=sub<G|x>;
> K:=sub<G|x^2,y>;
> H meet K;
Permutation group acting on a set of cardinality 4
Order = 2
    (1, 3)(2, 4)
> sub<G|H,K>;
Permutation group acting on a set of cardinality 4
    (1, 2, 3, 4)
    (1, 3)(2, 4)
    (1, 4)(2, 3)
Mapping from: GrpPerm: $, Degree 4 to GrpPerm: G
> L:=sub<G|H,K>;
> L eq G;
true
```

In the last construction of the subgroup generated by two subgroups, both a permutation group and a homomorphism were produced. This is not very important, and is to do with where Magma stores the resulting permutation group generated by the subgroups.

5.2 Producing Particular Subgroups

In a finite group, there are many concepts that are fundamental, like centralizers, normalizers, Sylow subgroups, and so on. In this section, we will learn how to construct these specific subgroups. We begin with normalizers, centralizers and normal closures.

```
> G:=Sym(4);
> x:=G!(1,2)(3,4);
> X:=sub<G|x>;
> Centralizer(G,x);
Permutation group acting on a set of cardinality 4
Order = 8 = 2^3
    (1, 3)(2, 4)
    (3, 4)
> NormalClosure(G,X);
Permutation group acting on a set of cardinality 4
Order = 4 = 2^2
    (1, 2)(3, 4)
    (1, 4)(2, 3)
> Normalizer(G,sub<G|(1,2)>);
Permutation group acting on a set of cardinality 4
Order = 4 = 2^2
    (3, 4)
    (1, 2)
> IsNormal(G,X);
false
> IsSubnormal(G,X);
true
```

Also of great importance are Sylow subgroups. The syntax is not difficult to guess. Here we also introduce the command PSL to produce the projective special linear group.

```
> G:=PSL(2,7);
> P:=SylowSubgroup(G,2);
> IsIsomorphic(P,DihedralGroup(4));
true Homomorphism of GrpPerm: P, Degree 8, Order 2^3 into GrpPerm: $, Degree 4,
Order 2^3 induced by
    (1, 7, 3, 8)(2, 6, 4, 5) |--> (1, 2, 3, 4)
    (1, 2)(3, 4)(5, 7)(6, 8) |--> (2, 4)
```

Constructing the derived subgroup and the core of a subgroup (largest normal subgroup contained within it) are also single-command actions.

```
> G:=SymmetricGroup(4);
> H:=SylowSubgroup(G,2);
> K:=Core(G,H);
> L:=DerivedSubgroup(G);
> L;
Permutation group L acting on a set of cardinality 4
Order = 12 = 2^2 * 3
    (1, 2, 3)
    (2, 3, 4)
> K;
Permutation group K acting on a set of cardinality 4
Order = 4 = 2^2
    (1, 2)(3, 4)
    (1, 4)(2, 3)
```

Finally, we look at commands to generate *all* subgroups or all normal subgroups.

```
> G:=SymmetricGroup(4);
> NormalSubgroups(G);
Conjugacy classes of subgroups
------------------------------


[1]     Order 1            Length 1
        Permutation group acting on a set of cardinality 4
        Order = 1
[2]     Order 4            Length 1
        Permutation group acting on a set of cardinality 4
        Order = 4 = 2^2
            (1, 4)(2, 3)
            (1, 3)(2, 4)
[3]     Order 12           Length 1
        Permutation group acting on a set of cardinality 4
        Order = 12 = 2^2 * 3
            (2, 3, 4)
            (1, 4)(2, 3)
            (1, 3)(2, 4)
[4]     Order 24            Length 1
        Permutation group acting on a set of cardinality 4
        Order = 24 = 2^3 * 3
            (3, 4)
            (2, 3, 4)
            (1, 4)(2, 3)
            (1, 3)(2, 4)
```

The information just produced is stored as a *record*. We have not met these before, but they are an important part of Magma's system of storing information. They are not accessed like sequences, but rather with using the left-quote.

```
> X:=NormalSubgroups(G);
> X[1];
rec<recformat<order, length, subgroup, presentation> |
    order := 1,
    length := 1,
    subgroup := Permutation group acting on a set of cardinality 4
    Order = 1
    >
> X[1,1];

>> X[1,1];
    ^
Runtime error in '[]': Bad argument types

> X[1]`order;
1
> H:=X[2]`subgroup;
> H;
Permutation group H acting on a set of cardinality 4
Order = 4 = 2^2
    (1, 4)(2, 3)
    (1, 3)(2, 4)
```

On may constrain precisely which subgroups appear by setting *parameters*. There are a wealth of possible parameters to attach here. In the example below we see an important one, namely

OrderEqual.

```
> G:=Sym(8);
> X:=Subgroups(G:OrderEqual:=8);
> Y:=[]; for i in X do if(IsTransitive(i`subgroup)) then Append(~Y,i`subgroup)\
; end if; end for;
> #Y;
5
> Y;
[
    Permutation group acting on a set of cardinality 8
    Order = 8 = 2^3
        (1, 4)(2, 8)(3, 5)(6, 7)
        (1, 5)(2, 6)(3, 4)(7, 8)
        (1, 6)(2, 5)(3, 8)(4, 7),
    Permutation group acting on a set of cardinality 8
    Order = 8 = 2^3
        (1, 2, 5, 6)(3, 8, 4, 7)
        (1, 3, 5, 4)(2, 7, 6, 8)
        (1, 5)(2, 6)(3, 4)(7, 8),
    Permutation group acting on a set of cardinality 8
    Order = 8 = 2^3
        (1, 8)(2, 3)(4, 6)(5, 7)
        (1, 3, 5, 4)(2, 7, 6, 8)
        (1, 5)(2, 6)(3, 4)(7, 8),
    Permutation group acting on a set of cardinality 8
    Order = 8 = 2^3
        (1, 5)(2, 6)(3, 8)(4, 7)
        (1, 4)(2, 8)(3, 5)(6, 7)
        (1, 2)(3, 7)(4, 8)(5, 6),
    Permutation group acting on a set of cardinality 8
    Order = 8 = 2^3
        (1, 8, 3, 4, 7, 6, 5, 2)
        (1, 5, 7, 3)(2, 6, 4, 8)
        (1, 7)(2, 4)(3, 5)(6, 8)
]
> Z:=Subgroups(G:OrderEqual:=8,IsTransitive:=true);
> #Z;
5
```

The possible parameters are given in the table below.

OrderEqual	OrderDividing	OrderMultipleOf
IndexLimit	IsElementaryAbelian	IsAbelian
IsCyclic	IsNilpotent	IsSolvable
IsNotSolvable	IsPerfect	IsRegular
	IsTransitive	

NOTE: this is the only time where you have to use the American version of 'solvable', rather than the British version of 'soluble'. The only parameter whose meaning is not clear is IndexLimit; the parameter `IndexLimit:=n` tells Magma to only find subgroups of index at most `n`.

To construct all maximal subgroups, simply use the command `MaximalSubgroups`. All of the parameters that may be added with the `Subgroups` command may be added to this command as well.

There are many other commands, and some of the more important ones are given in the table at the end of Section 5.6.

Exercise 5.1 Let G be a subgroup of A_7, such that $|G|$ is a multiple of 14. Prove that $G \cong A_7$ or $G \cong \mathrm{PSL}_2(7)$.

Exercise 5.2 How many nilpotent subgroups of S_6 are there?

Exercise 5.3 Construct the quaternion group Q_8. (This is the non-abelian group of order 8 that is not dihedral.) Prove that the central products $D_8 * D_8$ and $Q_8 * Q_8$ are isomorphic.

5.3 Conjugacy

Conjugacy is an integral part of group theory. Consequently, it is well-implemented in Magma. Elements and subgroups can be conjugated using the carat symbol:

```
> x:=Sym(4)!(1,4)
> y:=Sym(4)!(1,2,3);
> H:=sub<Sym(4)|[(1,2),(1,3)(2,4)]>;
> y^x;
(2, 3, 4)
> H^y;
Permutation group acting on a set of cardinality 4
Order = 8 = 2^3
    (2, 3)
    (1, 2)(3, 4)
```

This produces the conjugate of one element or subgroup by an element. To test whether two elements are conjugate, one needs to provide the two elements themselves and the group in which one wants to test conjugacy. If the two elements are conjugate, Magma returns a conjugating element as well as `true`.

```
> G:=Sym(5);
> H:=Alt(5);
> x:=G!(1,2,3,4,5);
> y:=G!(1,2,3,5,4);
> IsConjugate(G,x,y);
true (4, 5)
> IsConjugate(H,x,y);
false
```

Given an element and a group, one may calculate the conjugacy class containing the element.

```
> G:=DihedralGroup(8);
> Class(G,G!(1,7)(2,6)(3,5));
{
    (1, 7)(2, 6)(3, 5),
    (2, 8)(3, 7)(4, 6),
    (1, 5)(2, 4)(6, 8),
    (1, 3)(4, 8)(5, 7)
}
```

All conjugacy classes of a group may also be calculated, using an obvious command.

```
> G:=PSL(2,9);
> ConjugacyClasses(G);
Conjugacy Classes of group G
----------------------------
[1]     Order 1       Length 1
        Rep Id(G)


[2]     Order 2       Length 45
        Rep (1, 9)(2, 8)(3, 7)(4, 6)


[3]     Order 3       Length 40
        Rep (1, 9, 2)(3, 10, 6)(4, 7, 5)


[4]     Order 3       Length 40
```

```
        Rep (1, 4, 5)(3, 9, 10)(6, 8, 7)


[5]     Order 4       Length 90
        Rep (1, 8, 9, 2)(3, 6, 7, 4)


[6]     Order 5       Length 72
        Rep (1, 7, 6, 9, 2)(3, 5, 10, 4, 8)


[7]     Order 5       Length 72
        Rep (1, 6, 2, 7, 9)(3, 10, 8, 5, 4)
```

All of this information is not that difficult for Magma to compute, although for very large groups, it might be quite a lot of information in itself. Here we construct the Mathieu group M_{12}.

```
> H:=Sym(12);
> G:=sub<H|(1,4)(3,10)(5,11)(6,12),(1,8,9)(2,3,4)(5,12,11)(6,10,7)>;
> X:=ConjugacyClasses(G);
> #X;
15
> X[2];
<2, 396, (1, 10)(2, 12)(3, 7)(4, 11)(5, 6)(8, 9)>
> X[3,3];
(3, 11)(4, 6)(5, 9)(7, 12)
```

The last function to be described here is one that counts the number of conjugacy classes. This function is quicker than constructing all conjugacy classes. We demonstrate it with the Tits group, the derived subgroup of ${}^2F_4(2)$.

```
> G:=TitsGroup();
> NumberOfClasses(G);
22
> Order(G);
17971200
```

Exercise 5.4 The group $G = \mathrm{PSL}_2(11)$ has a Sylow 2-subgroup P isomorphic with V_4. Which of the elements of P are G-conjugate?

5.4 The Datatypes of Groups

All of the groups that we have dealt with so far are permutation groups. However, this is only one of the different types of group stored in Magma. They are, broadly speaking: permutation groups; matrix groups; abelian groups; finite soluble groups given by power-conjugate presentations; polycyclic groups; and finitely presented groups. Magma has methods that work in some of these cases and not in others. For example, it is obvious that `SylowSubgroup` is unlikely to work given an arbitrary finitely presented group, and in fact this method requires that the group is either an abelian group, a permutation group, a finite soluble group with a PC-presentation, or a matrix group. (We will collect a table of methods, together with the types of group with which they work, later.

We begin with permutation groups and matrix groups.

```
> G:=Sym(4);
> Type(G);
GrpPerm
> G:=SL(3,5);
> Type(G);
GrpMat
> Q:=Rationals();
> G:=GL(3,Q);
> Type(G);
GrpMat
```

A finite abelian group can be produced very easily in Magma, using a sequence of non-negative integers.

```
> AbelianGroup([2,2,5,4]);
Abelian Group isomorphic to Z/2 + Z/2 + Z/20
Defined on 4 generators
Relations:
    2*$.1 = 0
    2*$.2 = 0
    5*$.3 = 0
    4*$.4 = 0
> G<a,b,c,d>:=AbelianGroup([2,2,5,4]);
> G;
Abelian Group isomorphic to Z/2 + Z/2 + Z/20
Defined on 4 generators
Relations:
    2*a = 0
    2*b = 0
    5*c = 0
    4*d = 0
```

```
> 3*a;
a
> Type(G);
GrpAb
```

Like permutation groups have type `GrpPerm` and matrix groups have type `GrpMat`, abelian groups have type `GrpAb`. To force the particular abelian group constructed to be a different type of group, an extra argument is inserted.

```
> G<a,b,c,d>:=AbelianGroup(GrpPerm,[2,2,5,4]);
> G;
Permutation group G acting on a set of cardinality 13
Order = 80 = 2^4 * 5
    (1, 2)
    (3, 4)
    (5, 6, 7, 8, 9)
    (10, 11, 12, 13)
> Type(G);
GrpPerm
```

This argument can take the values `GrpPerm`, `GrpAb`, `GrpPC`, `GrpGPC`, or `GrpFP`, the last three of which we have not yet discussed. To construct an infinite abelian group, we must choose either the type `GrpFP` or the type `GrpGPC`.

```
> G<a,b,c,d>:=AbelianGroup(GrpPC,[0,2,5,4]);


>> G<a,b,c,d>:=AbelianGroup(GrpPC,[0,2,5,4]);
                               ^

Runtime error in 'AbelianGroup': Should be a sequence of small positive integers


> G<a,b,c,d>:=AbelianGroup(GrpFP,[0,2,5,4]);
> G;
Finitely presented group G on 4 generators
Relations
    b^2 = Id(G)
    c^5 = Id(G)
    d^4 = Id(G)
    a * b = b * a
    a * c = c * a
    a * d = d * a
    b * c = c * b
    b * d = d * b
    c * d = d * c
> G<a,b,c,d>:=AbelianGroup(GrpGPC,[0,2,5,4]);
> G;
GrpGPC : G of infinite order on 4 PC-generators
PC-Relations:
    b^2 = Id(G),
    c^5 = Id(G),
    d^4 = Id(G)
```

The next group type is GrpFP, or finitely presented group. These can be defined very easily; here we work with a presentation of Janko's smallest sporadic simple group, J_1.

```
> G<x,y>:=Group<x,y|x^2,y^3,(x*y)^7,(x*y*(x*y*x*y^-1)^3)^5,
> (x*y*(x*y*x*y^-1)^6*x*y*x*y*(x*y^-1)^2)^2>;
> Order(G);
175560
```

To identify this group properly, we need to turn it into a permutation group, so that we may analyze it. In order to do that, we pick a subgroup as large as we may easily find (here there is an obvious subgroup of order 7) and take the permutation representation of G on that subgroup. The commands for this are not difficult.

```
> H:=sub<G|x*y>;
> X:=CosetImage(G,H);
> CompositionFactors(X);
    G
    |  J1
    1
```

The command CompositionFactors has the obvious effect of calculating the group's composition factors.

Other groups may also be converted into finitely presented groups using the command FPGroup. We demonstrate here with the Mathieu group M_{11}, which we will feed Magma as a permutation group.

```
> N<a,b>:=PermutationGroup<11|(2,10)(4,11)(5,7)(8,9),(1,4,3,8)(2,5,6,9)>;
> CompositionFactors(N);
    G
    |  M11
    1
> G<x,y>:=FPGroup(N);
> G;
Finitely presented group G on 2 generators
Relations
    x^2 = Id(G)
    y^4 = Id(G)
    y^-1 * x * y^-2 * x * y^-2 * x * y^2 * x * y^2 * x * y^2 * x * y^-1 = Id(G)
    y * x * y * x * y^-1 * x * y * x * y^-2 * x * y^-1 * x * y * x * y^-1 * x *
    y^-1 * x = Id(G)
    y^-1 * x * y^-2 * x * y * x * y^-1 * x * y^-2 * x * y * x * y^-1 * x * y^2 *
    x * y * x = Id(G)
    (x * y^-1)^11 = Id(G)
```

Infinite, finitely presented, groups may be defined in Magma, although the functions available are slim. Here we have a look at a Baumslag–Solitar group.

```
> G<a,b>:=Group<a,b|a^b=a^2>;
> DerivedSubgroup(G);

>> DerivedSubgroup(G);
                  ^

Runtime error in 'DerivedSubgroup': Abelian quotient is too large

> AbelianQuotient(G);
Abelian Group isomorphic to Z
Defined on 1 generator (free)
> IsSoluble(G);

>> IsSoluble(G);
            ^

Runtime error in 'IsSoluble': Bad argument types
Argument types given: GrpFP

> SolubleQuotient(G);
GrpPC of order 1
PC-Relations:
Homomorphism of GrpFP: G into GrpPC induced by
    G.1 |--> Id($)
    G.2 |--> Id($)
[]
Soluble quotient terminates when free abelian section found.
```

```
> ncl<G|(a,b)>;

>> ncl<G|(a,b)>;
      ^

Runtime error in ncl< ... >: Could not construct a closed coset table
```

[Here the command `ncl` should produce the normal closure of a group, but it might fail in infinite cases.]

Here we see that you have to be careful when using an infinite, finitely presented group, as although the group can be shown to be soluble by the command `SolubleQuotient` returning the trivial subgroup, the command `IsSoluble` does not work.

It should also be mentioned that the Schreier method and methods regarding Tietze transformations have been implemented in Magma for working with finitely presented groups. The on-line help can provide more information.

The last two classes of groups to deal with are the polycyclic groups, of types `GrpGPC` and `GrpPC`. These require specific types of presentation, which will be discussed in the next section.

Exercise 5.5 The group Q_8 may be presented as

$$\langle\, a, b \,:\, a^4 = 1,\, a^2 = b^2,\, b^{-1}ab = a^{-1} \rangle.$$

Prove that this group is isomorphic with the group constructed in Exercise 1.

5.5 Power-Conjugate and Polycyclic Presentations

A power-conjugate (or power-commutator) presentation is a particular type of presentation for a soluble group, using the fact that a finite soluble group has a subnormal series all of whose factors are cyclic of order p. This type of presentation allows for (very!) efficient calculations within a soluble group, and most notably p-groups, and really should be used whenever a lot of work is to be done with a soluble group.

Suppose that G is a finite soluble group, and let

$$1 = G_{n+1} \trianglelefteq G_n \trianglelefteq \cdots \trianglelefteq G_1 = G$$

be a subnormal series for G such that G_i/G_{i+1} has prime order. Choose x_i to be an element of $G_i \setminus G_{i+1}$. Then there is a prime p (equal to $|G_i : G_{i+1}|$) such that $x_i^p \in \langle x_{i+1}, \dots, x_n\rangle$, and $x_n^p = 1$ in the case where $i = n$. Furthermore, it is also true that $x_j^{x_i}$ (where $i < j$) is a word in the generators $x_{i+1}, \dots, x_n$. (The G_i-normal closure of G_j is definitely contained within G_{i+1}.) The group G may be presented as

$$G = \langle\, x_1, \dots, x_n \,:\, x_i^{p_i} = w_{i,i},\; x_j^{x_i} = w_{i,j},\;\; 1 \leqslant i < j \leqslant n \rangle,$$

where the $w_{i,j}$ are words as described above. This is called a *power-conjugate presentation* of the group G. If, instead of conjugates, one used commutators, then this is called a *power-commutator presentation*. (Again, $[x_i, x_j]$ lives inside G_{i+1} since G_i/G_{i+1} is abelian.)

As we have said, groups given by a power-conjugate presentation are a much better type of group to work with than standard groups. The code for converting a given group into a power-conjugate group is given below, along with some evidence for it being a more efficient data type.

```
> G:=PermutationGroup<81|
> ( 1,14,27)( 2,15,25)( 3,13,26)( 4,17,21)( 5,18,19)( 6,16,20)
> ( 7,11,24)( 8,12,22)( 9,10,23)(28,32,36)(29,33,34)(30,31,35)(37,38,39)
> (46,47,48)(55,59,63)(56,60,61)(57,58,62)(64,65,66)(73,74,75),
> ( 1,41,81)( 2,42,79)( 3,40,80)( 4,44,75)( 5,45,73)( 6,43,74)
> ( 7,38,78)( 8,39,76)( 9,37,77)(10,50,63)(11,51,61)(12,49,62)(13,53,57)
> (14,54,55)(15,52,56)(16,47,60)(17,48,58)(18,46,59)(19,32,72)(20,33,70)
> (21,31,71)(22,35,66)(23,36,64)(24,34,65)(25,29,69)(26,30,67)(27,28,68)>;
> time H:=PCGroup(G);
Time: 0.020
> Order(H);
94143178827
> Ilog(3,Order(H));
23
> time Z:=Subgroups(H:OrderEqual:=3^20);
Time: 15.140
> time Z2:=Subgroups(G:OrderEqual:=3^20);
Time: 40.010
```

When asked to give the PC presentation of a PC group, Magma takes some shortcuts, which we will demonstrate now.

```
> G:=DihedralGroup(4);
> H:=PCGroup(G);
> H;
GrpPC : H of order 8 = 2^3
PC-Relations:
    H.2^2 = H.3,
    H.2^H.1 = H.2 * H.3
> H.1^2;
Id(H)
> H.3^2;
Id(H)
```

```
> H.3^H.1;
H.3
> H.3^H.2;
H.3
```

The extra equations after the line `H;` are those parts of the presentation omitted from Magma's output. Thus if a power relation is missing, it is that the power relation is $x^p = 1$ (note that this is only true for p-groups), and if a conjugate relation is missing, then it is assumed that the generators commute. The only exception to this is elementary abelian p-groups, which otherwise would have no relations at all!

```
> PCGroup(AbelianGroup([3,3,3]));
GrpPC of order 27 = 3^3
PC-Relations:
    $.1^3 = Id($),
    $.2^3 = Id($),
    $.3^3 = Id($)
```

The order of the generators is important in PC presentations.

```
> PolycyclicGroup<a,b|b^2,a^2=b>;
GrpPC of order 4 = 2^2
PC-Relations:
    $.1^2 = $.2
Mapping from: GrpFP to GrpPC
> PolycyclicGroup<b,a|b^2,a^2=b>;

>> PolycyclicGroup<b,a|b^2,a^2=b>;
                  ^
Runtime error in PolycyclicGroup< ... >: RHS not in collected form : $.2^2 = $.1
```

It is my recommendation that you define your group is some other way first, and then compute a PC presentation for it with the command `PCGroup`.

Polycyclic groups, the final type of group, are constructed similarly to groups with a power-conjugate presentation. The differences are that the exponents in the power relations need not be primes, and need not be there at all.

```
> G:=PolycyclicGroup<a,b|a^2,b^a=b^-1>;
> G;
GrpGPC : G of infinite order on 2 PC-generators
PC-Relations:
    G.1^2 = Id(G),
    G.2^G.1 = G.2^-1
```

The commands available for polycyclic groups are slim, since this is a new area of computational group theory. If such information is required, the help files contain much more information.

Extended Exercise 5.6 In this exercise we introduce the `SmallGroup` command. This command is your friend. The SmallGroup database contains complete information for all groups of order at most 2000 (except for 1024) and numerous other group orders.

(i) The number of groups in the SmallGroup database can be accessed with the command `NumberOfSmallGroups`. How many 101-groups of order 101^i are there for $2 \leqslant i \leqslant 7$? (Note that the number of groups of order p^8 is not yet known.)

(ii) How many of the groups of order 16 have exponent 4?

(iii) Find two 2-groups G and H such that the number of elements of each order in G and H are the same.

5.6 More Hard-Coded Groups and Commands

This section serves as a repository of the possible matrix and other groups that are given specific functions in Magma, that we haven't already described.

Helpfully, extraspecial groups are built into Magma.

```
> G:=ExtraSpecialGroup(3,1: Type:="+");
> Exponent(G);
3
> G:=ExtraSpecialGroup(3,1: Type:="-");
> Exponent(G);
9
```

The following table contains all of the classical groups.

Standard group	Projective version	Notes
`GL(n,q)`	`PGL(n,q)`	The groups $\mathrm{GL}_n(q)$ and $\mathrm{PGL}_n(q)$
`SL(n,q)`	`PSL(n,q)`	The groups $\mathrm{SL}_n(q)$ and $\mathrm{PSL}_n(q)$
`GU(n,q)`	`PGU(n,q)`	The groups $\mathrm{GU}_n(q)$ and $\mathrm{PGU}_n(q)$
`SU(n,q)`	`PSU(n,q)`	The groups $\mathrm{SU}_n(q)$ and $\mathrm{PSU}_n(q)$
`Sp(n,q)`	`PSp(n,q)`	The groups $\mathrm{Sp}_n(q)$ and $\mathrm{PSp}_n(q)$, n even
`Omega(n,q)`	`POmega(n,q)`	Simple orthogonal group, n odd
`SO(n,q)`	`PSO(n,q)`	Determinant 1 orthogonal group, n odd
`GO(n,q)`	`PGO(n,q)`	Full orthogonal group, n odd
`OmegaPlus(n,q)`	`POmegaPlus(n,q)`	Simple orthogonal group, plus type, n even
`SOPlus(n,q)`	`PSOPlus(n,q)`	Determinant 1 orthogonal group, plus type, n even
`GOPlus(n,q)`	`PGOPlus(n,q)`	Full orthogonal group, plus type, n even
`OmegaMinus(n,q)`	`POmegaMinus(n,q)`	Simple orthogonal group, minus type, n even
`SOMinus(n,q)`	`PSOMinus(n,q)`	Determinant 1 orthogonal group, minus type, n even
`GOMinus(n,q)`	`PGOMinus(n,q)`	Full orthogonal group, minus type, n even

In addition, the commands `SuzukiGroup(q)` and `ReeGroup(q)` produce the Suzuki and small Ree groups respectively, if q is the correct power of the correct prime. Some more matrix groups were introduced in Version 2.14 of Magma, such as the large Ree group, and conformal symplectic and unitary groups, although we will not discuss them here.

In fact, any matrix group of Lie type can be constructed, using the generic `ChevalleyGroup` function. Here is an example, creating the matrix group $G_2(7)$.

```
> ChevalleyGroup("G",2,7);
MatrixGroup(7, GF(7))
Generators:
    [3 0 0 0 0 0 0]
    [0 5 0 0 0 0 0]
    [0 0 2 0 0 0 0]
    [0 0 0 1 0 0 0]
    [0 0 0 0 4 0 0]
    [0 0 0 0 0 3 0]
    [0 0 0 0 0 0 5]

    [6 0 1 0 0 0 0]
    [6 0 0 0 0 0 0]
    [0 6 0 6 0 1 0]
    [0 5 0 6 0 0 0]
    [0 6 0 0 0 0 0]
    [0 0 0 0 1 0 1]
    [0 0 0 0 1 0 0]
```

The first value of the `ChevalleyGroup` function is the Lie type of the group. This can be one of `A`, `B`, `C`, `D`, `E`, `F`, `G`, `2A`, `2B`, `2D`, `3D`, `2E`, `2F`, and `2G`. The second value is the integer label of the series, which can be any integer for `A`, and so on. The final value is the value q of the finite field, which normally may be arbitrary, except in some twisted groups.

This normally produces the natural representation, unless the group is $G_2(2^n)$, in which case it produces the 7-dimensional representation. Magma can be forced in this case to produce the 6-dimensional representation, by using a parameter.

```
> G<x,y>:=ChevalleyGroup("G",2,4);
> x;
[  $.1      0      0      0      0      0      0]
[     0 $.1^2      0      0      0      0      0]
[     0      0 $.1^2      0      0      0      0]
[     0      0      0      1      0      0      0]
[     0      0      0      0    $.1      0      0]
[     0      0      0      0      0    $.1      0]
[     0      0      0      0      0      0 $.1^2]
> H<a,b>:=ChevalleyGroup("G",2,4:Irreducible:=true);
> a;
[  $.1      0      0      0      0      0]
[     0 $.1^2      0      0      0      0]
[     0      0 $.1^2      0      0      0]
[     0      0      0    $.1      0      0]
[     0      0      0      0    $.1      0]
[     0      0      0      0      0 $.1^2]
```

This produces all simple groups of Lie type, apart from the Tits group, which can be returned by `TitsGroup()`, as we have seen before.

Having considered groups, we now consider commands. In addition to `GrpPerm`, the following commands accept the following group types.

Command	Domain
`Centre(G)`	GrpMat, GrpAb, GrpPC, GrpGPC
`Hypercentre(G)`	GrpAb, GrpPC
`DerivedSubgroup(G)`	GrpMat, GrpAb, GrpPC, GrpGPC, GrpFP
`DerivedLength(G)`	GrpMat, GrpAb, GrpPC, GrpGPC
`DerivedSeries(G)`	GrpMat, GrpAb, GrpPC, GrpGPC
`FittingSubgroup(G)`	GrpAb, GrpPC, GrpGPC
`FrattiniSubgroup(G)`	GrpAb, GrpPC, GrpGPC
`JenningsSeries(G)`	GrpMat, GrpPC
`LowerCentralSeries(G)`	GrpMat, GrpPC, GrpGPC
`UpperCentralSeries(G)`	GrpAb, GrpMat, GrpPC, GrpGPC
`NilpotencyClass(G)`	GrpMat, GrpAb, GrpPC, GrpGPC
`pCentralSeries(G)`	GrpMat, GrpPC
`Radical(G)`	GrpMat
`SolubleRadical(G)`	GrpMat, GrpPC
`SolubleResidual(G)`	GrpMat
`SubnormalSeries(G,H)`	GrpMat, GrpAb, GrpPC
`CompositionFactors(G)`	GrpMat, GrpAb, GrpPC
`SylowSubgroup(G,p)`	GrpMat, GrpAb, GrpPC

5.7 Extensions and Automorphisms

Magma has functions to deal with extensions of groups, and automorphism groups. The easiest one is the function `AutomorphismGroup`, which strangely enough constructs the automorphism group of a finite group. This group is of type `GrpAuto`, and not a great deal may be done with this category. One thing that can be done is to construct the automorphism group as a permutation group on the non-trivial elements of the group. The following example demonstrates some of the limitations of the `GrpAuto` type.

```
> G:=AbelianGroup(GrpPerm,[3,3]);
> A:=AutomorphismGroup(G);
> IsIsomorphic(A,GL(2,3));


>> IsIsomorphic(A,GL(2,3));
               ^
Runtime error in 'IsIsomorphic': Bad argument types
Argument types given: GrpAuto, GrpMat


> Order(A);
```

```
48
> H:=PermutationGroup(A);
> IsIsomorphic(H,GL(2,3));
true Homomorphism of GrpPerm: H, Degree 8, Order 2^4 * 3 into GL(2, GF(3)) induced
by
    Id(H) |--> [1 0]
    [0 1]
    Id(H) |--> [1 0]
    [0 1]
    (1, 2)(3, 5)(4, 7) |--> [2 2]
    [0 1]
    (1, 3, 6)(2, 4, 8) |--> [1 0]
    [2 1]
```

If `A` is an automorphism group, then `A.i` returns the `i`th generator of `A`, and one may test whether an automorphism is inner with the command `IsInner`.

In this next example, we prove that the two conjugacy classes of subgroups of index 12 in the Mathieu group M_{12} are conjugate via an outer automorphism.

```
> S:=Sym(12);
> G:=sub<S|(1,4)(3,10)(5,11)(6,12),(1,8,9)(2,3,4)(5,12,11)(6,10,7)>;
> H:=Stabilizer(G,12);
> A:=AutomorphismGroup(G);
> for i in Generators(A) do
for> IsConjugate(G,H,H@i); end for;
true Id(G)
true (1, 12, 3, 2)(5, 6, 11, 7)
true (1, 12, 3, 2)(5, 6, 11, 7)
true (1, 12, 5, 8, 11, 6, 7, 9, 4, 3, 2)
true Id(G)
true (1, 12, 9, 7, 11, 4, 3, 2)(5, 6, 8, 10)
true (1, 12, 2)(5, 7, 9)(6, 10, 11)
true (1, 12, 7, 4, 3, 2)(5, 9, 10, 8, 6, 11)
true (1, 12, 11, 7, 10, 9, 8, 5, 4, 3, 2)
false
> for i in Generators(A) do IsInner(i); end for;
true (1, 2, 3, 7)(5, 11, 8, 9)
true (2, 7, 9)(3, 4, 12)(5, 11, 8)
true (1, 10)(2, 9)(3, 12)(5, 8)
true (1, 10, 9)(3, 8, 4)(5, 11, 12)
true (1, 9, 3)(2, 7, 11)(4, 10, 8)
true (1, 2)(3, 5)(4, 11)(6, 8)(7, 10)(9, 12)
true (1, 8)(2, 12)(7, 10)(9, 11)
true (1, 9, 10, 4, 3)(2, 12, 7, 11, 5)
true (1, 7, 2)(3, 8, 5)(4, 12, 11)
false
```

Knowing about automorphisms of groups is all well and good, but we would like to also produce split extensions of groups by automorphisms. This can be done with `Holomorph`.

```
> X:=AbelianGroup(GrpPerm,[2,2]);
> IsIsomorphic(Sym(4),Holomorph(X));
true Homomorphism of GrpPerm: $, Degree 4, Order 2^3 * 3 into GrpPerm: $, Degree 4,
Order 2^3 * 3 induced by
    (1, 2, 3, 4) |--> (1, 3, 2, 4)
    (1, 2) |--> (1, 3)
```

Semidirect products of groups with subgroups of their automorphism groups can be achieved by adding an extra value in the command. Here we will construct A_4 as the semidirect product of V_4 by an outer automorphism of order 3.

```
> X:=AbelianGroup(GrpPerm,[2,2]);
> A:=AutomorphismGroup(X);
> NumberOfGenerators(A);
4
> for i in [1..4] do Order(A.i); end for;
1
1
2
2
> x:=A.3*A.4;
> H:=sub<A|x>;
> G:=Holomorph(X,H);
> G;
Permutation group G acting on a set of cardinality 4
    (1, 2)(3, 4)
    (1, 3)(2, 4)
    (2, 4, 3)
> IsIsomorphic(G,Alt(4));
true Homomorphism of GrpPerm: G, Degree 4, Order 2^2 * 3 into GrpPerm: $, Degree 4,
Order 2^2 * 3 induced by
    (1, 2)(3, 4) |--> (1, 4)(2, 3)
    (1, 3)(2, 4) |--> (1, 3)(2, 4)
    (2, 4, 3) |--> (2, 3, 4)
```

If you only need a direct product, rather than a semidirect product, you simply use the command `DirectProduct`. This either takes a pair of groups (with the same type) or a sequence of groups.

```
> G:=DirectProduct([CyclicGroup(2):i in [1..4]]);
> Order(G);
16
```

The wreath product of G and H can be constructed using `WreathProduct(G,H)`.

We will now briefly consider cohomology. Recently, the implementation of this has been significantly improved, and we will discuss only the new version. (For backwards compatibility, the old functions are still available.) Normally this requires a module for the group, but here we use the trivial module. After the next chapter, we will be able to use other modules.

We begin by defining the trivial module. Given the trivial module (or indeed any module), one forms the *cohomology module*, with which all calculations are made. For example, the 0th, 1st and 2nd cohomology groups are calculable.

```
> G:=PSL(2,7);
> k:=TrivialModule(G,GF(2));
> M:=CohomologyModule(G,k);
> CohomologyGroup(M,0);
Full Vector space of degree 1 over GF(2)
> CohomologyGroup(M,1);
Full Vector space of degree 0 over GF(2)
> CohomologyGroup(M,2);
Full Vector space of degree 1 over GF(2)
```

It is actually possible to work with cocycles themselves, but this is beyond the scope of this course. If more about cohomology is needed, then the Magma on-line help files are very useful.

Chapter 6

Representation Theory

Now that we have a reasonable amount of information on how to deal with groups, let's now have a look at representations.

6.1 Character Theory

We begin with character theory, since this is the easiest to do. To construct the character table of a group, one simple uses the obvious function.

```
> G:=AlternatingGroup(5);
> CharacterTable(G);


Character Table of Group G
--------------------------



---------------------------
Class |   1  2  3    4    5
Size  |   1 15 20   12   12
Order |   1  2  3    5    5
---------------------------
p  =  2   1  1  3    5    4
p  =  3   1  2  1    5    4
p  =  5   1  2  3    1    1
---------------------------
X.1   +   1  1  1    1    1
X.2   +   3 -1  0   Z1 Z1#2
X.3   +   3 -1  0 Z1#2   Z1
```

```
X.4   +   4  0  1   -1   -1
X.5   +   5  1 -1    0    0


Explanation of Character Value Symbols
--------------------------------------

# denotes algebraic conjugation, that is,
#k indicates replacing the root of unity w by w^k

Z1     = (CyclotomicField(5: Sparse := true)) ! [ RationalField() | 1, 0, 1, 1 ]



[]
```

Using our previous knowledge of cyclotomic fields, we can understand the irrationality occurring here.

```
> F<a>:=CyclotomicField(5:Sparse:=true);
> Z1:=F![ RationalField() | 1, 0, 1, 1 ];
> Z1;
a^3 + a^2 + 1
> MinimalPolynomial(Z1);
$.1^2 - $.1 - 1
```

Specifying an individual character is easy.

```
> G:=Alt(5);
> X:=CharacterTable(G);
> x1:=X[2];
> x2:=X[3];
> x1*x2;
( 9, 1, 0, -1, -1 )
> InnerProduct(x1*x2,X[1]);
0
> InnerProduct(x1*x2,X[4]);
1
```

The Frobenius–Schur indicator, although visible on the character table on the left-hand side, is accessible through `Indicator(x)`. Given a character, the commands `IsIrreducible` and `IsFaithful` have obvious meanings, and `IsReal` test whether the character is real (not the representation).

Important concepts in character theory are the induction and restriction of characters and the construction of permutation characters. Given a group G, a subgroup H and characters χ and ϕ of G and H respectively, one may easily perform these operations.

```
> G<x,y>:=PermutationGroup<23|\[
> 2,1,4,3,5,6,8,7,10,9,11,12,14,13,16,15,17,18,20,19,22,21,23]
> ,\[
> 16,9,1,5,8,22,7,23,21,10,3,2,20,18,17,11,15,6,19,13,12,14,4]
> >;
> CompositionFactors(G);
    G
    |  M23
    1
> Z:=MaximalSubgroups(G);
> #Z;
7
> H:=Z[4]`subgroup;
> CompositionFactors(H);
    G
    |  Alternating(8)
    1
> X1:=CharacterTable(G);
> X2:=CharacterTable(H);
> x:=PermutationCharacter(G,H);
> x;
( 506, 42, 11, 6, 1, 3, 2, 2, 0, 0, 0, 0, 0, 1, 1, 0, 0 )
> X1[9];
( 253, 13, 1, 1, -2, 1, 1, 1, -1, 0, 0, -1, -1, 1, 1, 0, 0 )
> Restriction(X1[9],H);
( 253, 13, 13, 1, 1, 1, 1, -2, 1, 1, 1, 1, 1, 1 )
> X2[14];
( 70, -2, 2, -5, 1, -2, 0, 0, -1, 1, 0, 0, 0, 0 )
> Induction(X2[14],G);
( 35420, 28, 5, -4, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 )
```

6.2 Constructing and Working with Modules

There are three main ways to construct modules: from a permutation representation on a subgroup; from the natural representation of a matrix group; and by explicitly giving the matrices. In the example below, we shall see all three of these ways, together with a new way of defining a matrix, called a `CambridgeMatrix`. The reason for using this type of matrix as an input is that it is a very efficient (in terms of space requirements) method of storing matrices. We will also see the commands `SocleFactors`, `CompositionFactors`, and `Tensor Product`, each of which has its obvious meaning.

```
> G:=PSL(3,4);
> M:=PermutationModule(G,GF(3));
> M;
GModule M of dimension 21 over GF(3)
> SocleFactors(M);
[
    GModule of dimension 1 over GF(3),
    GModule of dimension 19 over GF(3),
    GModule of dimension 1 over GF(3)
]
> H:=GL(3,4);
> N:=GModule(H);
> N;
GModule N of dimension 3 over GF(2^2)
> G<x,y>:=PermutationGroup<11|\[
> 1,10,3,11,7,6,5,9,8,2,4]
> ,\[
> 4,5,8,3,6,9,7,1,2,10,11]
> >;
> F:=GF(3);
> x:=CambridgeMatrix(1,F,5,[
> "02011",
> "21102",
> "11122",
> "02222",
> "02210"]);
>
> y:=CambridgeMatrix(1,F,5,[
> "01101",
> "20010",
> "00122",
> "21000",
> "01221"]);
> M2:=GModule(G,[x,y]);
> M2;
GModule M2 of dimension 5 over GF(3)
> CompositionFactors(TensorProduct(M2,M2));
[
    GModule of dimension 5 over GF(3),
    GModule of dimension 10 over GF(3),
    GModule of dimension 10 over GF(3)
]
```

One way of constructing all simple modules for a group G over a field K is to use the command `IrreducibleModules`. This works well for groups whose irreducible representations are of fairly small degree, but starts to get slow for groups where the simple modules have large degree (say in the thousands).

```
> G:=PSL(2,7);
> time X:=IrreducibleModules(G,GF(2));
Time: 0.040
> X;
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 3 over GF(2),
    GModule of dimension 3 over GF(2),
    GModule of dimension 8 over GF(2)
]
> H:=PSL(3,5);
> time Y:=IrreducibleModules(H,GF(2));
Time: 5.100
> Y;
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 30 over GF(2),
    GModule of dimension 124 over GF(2),
    GModule of dimension 124 over GF(2),
    GModule of dimension 480 over GF(2),
    GModule of dimension 480 over GF(2)
]
```

To compare, we look at the Higman–Sims simple group, which has irreducible modules of degree between one- and two-thousand.

```
> G<x,y>:=PermutationGroup<100|\[
> 60,72,81,43,11,87,34,8,63,10,5,46,28,71,42,97,17,57,52,20,32,22,47,54,83,
> 78,27,13,89,39,31,21,61,7,56,36,67,38,30,40,41,15,4,76,88,12,23,59,86,74,
> 66,19,99,24,75,35,18,58,48,1,33,73,9,64,79,51,37,82,69,70,14,2,62,50,55,
> 44,92,26,65,80,3,68,25,90,98,49,6,45,29,84,91,77,93,100,95,96,16,85,53,94]
> ,\[
> 86,53,40,29,98,27,83,38,23,47,42,14,10,81,90,20,3,69,59,28,22,64,89,6,17,
> 58,51,7,92,8,56,11,52,62,93,82,15,2,100,48,80,46,4,33,84,91,1,25,67,34,
> 73,94,30,65,68,37,63,96,79,74,9,87,71,39,5,54,41,12,45,49,99,36,24,72,44,
> 18,26,50,35,70,55,60,16,76,77,13,78,43,95,31,32,88,19,75,61,85,57,66,97,21]
> >;
> CompositionFactors(G);
    G
    |  HS
    1
> time X:=IrreducibleModules(G,GF(2));
Time: 44.730
> X;
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 20 over GF(2),
    GModule of dimension 56 over GF(2),
    GModule of dimension 132 over GF(2),
    GModule of dimension 518 over GF(2),
    GModule of dimension 1000 over GF(2),
    GModule of dimension 1408 over GF(2),
    GModule of dimension 1792 over GF(2)
]
```

There are various restrictions on which combinations of fields and groups are allowed: one should consult the Magma on-line help for details.

Finally, if one has a module for a group and wishes to induce or restrict it, one needs the commands `Induction` and `Restriction`. The dual of a module `M` is simply returned by `Dual(M)`. Of course, the permutation module for G on the cosets of some subgroup H is isomorphic to the induced trivial module from the subgroup H; the trivial module for a group `G` over a field `K` may be created using the command `TrivialModule(G,K)`.

Now that we have our modules, we want to be able to work with them. The tensor product of two modules, and the tensor powers of a module are easy to take.

```
> G:=PSL(2,7);
> X:=IrreducibleModules(G,GF(2));
> X;
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 3 over GF(2),
    GModule of dimension 3 over GF(2),
    GModule of dimension 8 over GF(2)
]
> M:=TensorProduct(X[2],X[3]);
> IndecomposableSummands(M);
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 8 over GF(2)
]
> N:=TensorPower(X[2],5);
> #CompositionFactors(N);
71
```

Similarly the commands `SymmetricSquare` and `ExteriorSquare`, and the more general functions `SymmetricPower` and `ExteriorPower`, are available.

The tools for analyzing module structure are quite powerful. The purpose of the commands `IndecomposableSummands` and `CompositionFactors` are obvious, and the commands `Socle` and `JacobsonRadical` also exist. The socle series is returned with `SocleSeries`, and the successive quotients in this series are returned with `SocleFactors`. To check whether a module is decomposable or irreducible, without decomposing or reducing it completely, the commands `IsDecomposable` and `IsIrreducible` are available. They return `true` or `false`, together with a submodule and its quotient if the answer is `false`. Note that decomposing modules is difficult.

To end, we must consider the direct sum of two (or more) modules, and how to break up modules that, while irreducible (or indecomposable) over one field, become reducible (or decomposable) over a larger field.

```
> G<x,y>:=PermutationGroup<23|\[
> 2,1,4,3,5,6,8,7,10,9,11,12,14,13,16,15,17,18,20,19,22,21,23]
> ,\[
> 16,9,1,5,8,22,7,23,21,10,3,2,20,18,17,11,15,6,19,13,12,14,4]>;
> CompositionFactors(G);
    G
    |  M23
    1
```

```
> X:=IrreducibleModules(G,GF(2));
> X;
[
    GModule of dimension 1 over GF(2),
    GModule of dimension 11 over GF(2),
    GModule of dimension 11 over GF(2),
    GModule of dimension 44 over GF(2),
    GModule of dimension 44 over GF(2),
    GModule of dimension 120 over GF(2),
    GModule of dimension 220 over GF(2),
    GModule of dimension 220 over GF(2),
    GModule of dimension 252 over GF(2),
    GModule of dimension 1792 over GF(2)
]
> M:=ChangeRing(X[#X],GF(4));
> IsIrreducible(M);
false GModule of dimension 896 over GF(2^2)
GModule of dimension 896 over GF(2^2)
> DirectSum(X[4],X[5]);
GModule of dimension 88 over GF(2)
> DirectSum(X);
GModule of dimension 2715 over GF(2)
```

The last command enables one to take the n-fold direct sum of a module, with a command like that below.

```
> DirectSum(M:i in [1..10]);
```

Exercise 6.1 Here we will implement an algorithm for removing the projective summands from a module for a p-group G. Suppose that G is a p-group, and let K be a field of characteristic p. Write KG for the group algebra, which is the smallest free module. (Note that it is also the trivial module induced from the trivial subgroup to the whole group; i.e., the permutation module of G on the cosets of the trivial subgroup.)

If M is a KG-module, and M contains a free submodule, then given a random element x of M, the subgroup generated by x will likely be free. Any free submodule is a summand, and so may be quotiented out. The process is then repeated until no more free summands may be found in this manner. (If twenty submodules are taken in this fashion and none is free, then the module may be assumed to have no free submodules.)

To take a random element in a module `M`, simply use `Random(M)`. The submodule generated by one element is produced by the function `sub<M|Random(M)>`. Finally, quotienting `M` by `N` is easy and given by M/N.

Create a function in Magma, called RemoveFreeSummands, which takes a module M and returns a module N which is isomorphic with M, but with all free summands removed. [You will need the following other commands: Group(M) and Field(M) return, respectively, the group and the field associated with M.]

6.3 Submodules and Quotient Modules

As we saw in Exercise 6.1, one may construct a submodule generated by an element or a sequence of elements with sub<M|X>, where M is the module and X is a collection of elements.

Constructing all submodules of a module is easy, using the command Submodules. There is an optional parameter, namely CodimensionLimit, which, as its name suggests, limits the codimension of the submodules found.

```
> G:=AbelianGroup(GrpPerm,[2,2]);
> K:=GF(2);
> K2:=GF(4);
> KG:=PermutationModule(G,sub<G|>,K);
> Submodules(KG);
[
    GModule of dimension 0 over GF(2),
    GModule of dimension 1 over GF(2),
    GModule of dimension 2 over GF(2),
    GModule of dimension 2 over GF(2),
    GModule of dimension 2 over GF(2),
    GModule of dimension 3 over GF(2),
    GModule KG of dimension 4 over GF(2)
]
> K2G:=PermutationModule(G,sub<G|>,K2);
> Submodules(K2G);
[
    GModule of dimension 0 over GF(2^2),
    GModule of dimension 1 over GF(2^2),
    GModule of dimension 2 over GF(2^2),
    GModule of dimension 2 over GF(2^2),
    GModule of dimension 2 over GF(2^2),
    GModule of dimension 2 over GF(2^2),
    GModule of dimension 2 over GF(2^2),
    GModule of dimension 3 over GF(2^2),
    GModule K2G of dimension 4 over GF(2^2)
]
> G:=AbelianGroup(GrpPerm,[3,3]);
> K:=GF(3);
> KG:=PermutationModule(G,sub<G|>,K);
> M:=DirectSum([KG:i in [1..3]]);
> time X:=Submodules(M:CodimensionLimit:=4);
Time: 311.220
> #X;
205064
> Dimension(M)-Dimension(X[1]);
4
> Dimension(M)-Dimension(X[#X]);
0
```

Given two submodules `N` and `N2` of a module `M`, the join and meet of these two submodules is given by `N+N2` and `N meet N2` respectively. As mentioned before, the quotient module of `M` by `N` is simple given by `M/N`. It is also possible to deal with the lattice of submodules as a lattice, but this is beyond the scope of the course.

The command `AHom` constructs all algebra homomorphisms from the domain to the codomain.

```
> G:=AbelianGroup(GrpPerm,[3,3]);
> K:=GF(3);
> KG:=PermutationModule(G,sub<G|>,K);
> Triv:=TrivialModule(G,K);
> A:=AHom(KG,Triv);
> A;
KMatrixSpace of 9 by 1 GHom matrices and dimension 1 over GF(3)
> Image(Random(A));
GModule Triv of dimension 1 over GF(3)
> SocleSeries(KG);
[
    GModule of dimension 1 over GF(3),
    GModule of dimension 3 over GF(3),
    GModule of dimension 6 over GF(3),
    GModule of dimension 8 over GF(3),
    GModule KG of dimension 9 over GF(3)
]

> x:=Random(A); IsIsomorphic(SocleSeries(KG)[4],Kernel(x));
```

```
true
```

In the next example we compare the random algorithm for choosing elements of the space of all algebra homomorphisms given by `AHom` with the actual distribution of elements.

```
> G:=AbelianGroup(GrpPerm,[3,3]);
> K:=GF(3);
> KG:=PermutationModule(G,sub<G|>,K);
> M:=KG/SocleSeries(KG)[3];
> Dimension(M);
3
> A:=AHom(KG,M);
> X:=[0,0,0,0];
> time for i in [1..10000] do
time|for> d:=Dimension(Image(Random(A)));
time|for> X[d+1]:=X[d+1]+1;
time|for> end for;
Time: 0.070
> X;
[ 380, 2921, 0, 6699 ]
> Y:=[0,0,0,0];
> time for i in A do
time|for> d:=Dimension(Image(i));
time|for> Y[d+1]:=Y[d+1]+1;
time|for> end for;
Time: 0.000
> Y;
[ 1, 8, 0, 18 ]
> [380*i : i in Y];
[ 380, 3040, 0, 6840 ]
```

Extended Exercise 6.2 This exercise will develop the concept of periodicity for modules over p-groups. The *core* of a module is the module with all projective summands removed. In the case of a p-group, all projective modules are free, so this is the module returned by the function developed in Exercise 6.1. The core of a module M is denoted by $\Omega^0(M)$.

Every module M has a projective cover P: in the case of a p-group, this is easy to describe. It is simply the free module consisting of n copies of KG, where n is the dimension of M modulo its Jacobson radical. There is a surjective homomorphism $P \to M$, whose kernel has no projective summands; this module is denoted by $\Omega(M)$. This sets up a bijection on the set of all modules M that are cores. Consequently, we may produce the function Ω^i, which is the iterated application of this function.

A module M is called *periodic* if, for some $i > 0$, we have $\Omega^i(M) = \Omega^0(M)$. According to a theorem of Carlson, a periodic module must have dimension a multiple of p^{r-1}, where r is the p-rank of G (rank of a largest elementary abelian subgroup of G). Let Q be a maximal abelian p-subgroup of smallest order. Then the period of a periodic module divides $2|G : Q|$.

(i) Write a program to calculate the p-rank of a finite group G.

(ii) Write a program to find a maximal abelian subgroup of a p-group G of smallest order.

(iii) Write a program to calculate, given a module M, the module $\Omega(M)$.

(iv) Write a program to test whether a module is periodic, and if so, find its period. This function should return either `false` or `true n`, where `n` is the period.

Appendix A

Managing Errors

In the previous chapters we have seen many examples of errors. It might be useful for you to create your own errors, and to manage errors that crop up. Since an error message terminates all procedures, you might want to 'try' a piece of code, and if an error occurs, 'catch' it instead of letting it terminate your process.

A.1 Throwing an Error

Suppose that we have a function, like the one created in Exercise 6.1, that takes a module for a group of order a prime power. We might want to make sure early on that we were actually given a module, and that the associated group is a p-group, and that the field is of the right characteristic! We can do this using error statements.

This would be done as follows. (Here we demonstrate two different ways of producing errors. The `error if` method is preferred.)

```
> function RemoveProjectiveSummands(M)
function> if Type(M) ne ModGrp then
function|if> error "Error in 'RemoveProjectiveSummands': Bad argument types";
function|if> end if;
function> G:=Group(M);
function> error if(not(IsPrimePower(Order(G)))),
function|error> "Error in 'RemoveProjectiveSummands': Group is not a p-group";
function> Bool,p:=IsPrimePower(Order(G));
function> K:=Field(M);
function> error if Characteristic(K) ne p,
function|error> "Error in 'RemoveProjectiveSummands': Characteristic does not \
match up with group";
function> KG:=PermutationModule(G,sub<G|>,K);
function> n:=0;
function> repeat n+:=1;
function|repeat> N:=sub<M|Random(M)>;
function|repeat> if(IsIsomorphic(N,KG)) then
function|repeat|if> M:=M/N;
function|repeat|if> n:=0;
function|repeat|if> end if;
function|repeat> until n eq 20;
function> return M;
function> end function;
```

This function is demonstrated below.

```
> G:=CyclicGroup(6);
> H:=AbelianGroup(GrpPerm,[3,3]);
> K:=GF(2);
> F:=GF(3);
> KG:=PermutationModule(G,sub<G|>,K);
> KH:=PermutationModule(H,sub<H|>,K);
> FH:=PermutationModule(H,sub<H|>,F);
> M:=DirectSum(FH,TrivialModule(H,F));
> RemoveProjectiveSummands(G);
Error in 'RemoveProjectiveSummands': Bad argument types
> RemoveProjectiveSummands(KG);
Error in 'RemoveProjectiveSummands': Group is not a p-group
> RemoveProjectiveSummands(KH);
Error in 'RemoveProjectiveSummands': Characteristic does not match up with group
> RemoveProjectiveSummands(M);
GModule of dimension 1 over GF(3)
```

A.2 Catching an Error

Now that we know how to throw errors, we need to know how to catch them as well, so that they don't ruin our day. This is accomplished using the `try` command. You catch the error is one occurs, and label it something, such as `e`. We will see what to do with `e` next, but first an example.

```
> Factorization([3]);


>> Factorization([3]);
                 ^

Runtime error in 'Factorization': Bad argument types
```

```
Argument types given: SeqEnum[RngIntElt]

> try
try> Factorization([3]);
try> catch e
try> print "This didn't work";
try> print "Should do something else here";
try> end try;
This didn't work
Should do something else here
```

The error is a record, and has four attributes: Traceback; Position; Object; and Type. Traceback is not helpful for us, and so we will ignore it. Position stores where the error occurred, Object stores the description of the error, or if it is a user error, the label of the error given (which was again a description above), and Type is either `Err` for a system error or `ErrUser` if it is a user-defined error, which we haven't defined here. For example, here is the result of the Position attribute.

```
> try
try> Factorization([3]);
try> catch e
try> e`Position;
try> end try;
>> Factorization([3]);
                ^
```

Here we see an error raised by a user program.

```
> function MyFunction(a);
function> error if a eq 0,
function|error> "Error here!";
function> return a+1;
function> end function;
> MyFunction(0);
Error here!
> try MyFunction(0);
try> catch e
try> e`Object;
try> e`Type;
try> end try;
Error here!
Err
```

Appendix B

More on Functions

In this appendix, we will see some of the more advanced topics on using functions, namely: creating functions with an arbitrary number of inputs; creating parameters for functions; and integrating your functions into Magma.

B.1 Variadic Functions and Parameters

Variadic functions are Magma's name for functions that take an unspecified number of inputs. To construct a variadic function, an ellipsis is placed after the final variable.

```
> function MyFunction(a,b,...)
function> return b;
function> end function;
> MyFunction(2,3);
[* 3*]
> MyFunction(2,3,"Ham",[4]);
[* 3, Ham,
    [ 4 ]
*]
```

In this example, it becomes obvious what Magma does: it keeps the first $n-1$ variables as they are and assigns to the final variable's name a *list*, containing all subsequent inputs. With a bit of thought, it should be possible to design your own variadic functions when you need them.

Parameters are easy to define. The first line of a function with parameters should look like this.

```
function MyFunction(n,m: Param1:=true, Param2:=20)
```

At some point in the code, it might be useful to check that the parameter values are actually meaningful, throwing an error if they are not. In the next section we will introduce intrinsics, where the error-checking of parameters is much easier. However, if you are defining a function, it might be useful to do some checking.

B.2 Intrinsics and Packages

Intrinsics are the last step in fully integrating your function into Magma. They may only be used inside *packages*, which are special types of text files that provide functions. We turn our example function in Appendix A into an intrinsic. This is just like a function, except with some extra paraphernalia. It includes the types of the inputs, and contains a comment at the start so that one may get information on the purpose of the function in the same way as with Magma's other functions.

```
intrinsic RemoveFreeSummands(M::ModGrp) -> ModGrp
{Removes all free summands from a module for a p-group. Returns an error if the
group is not a p-group and the field's characteristic is not also p.}

G:=Group(M);

error if(not(IsPrimePower(Order(G)))),
  "Error in 'RemoveProjectiveSummands': Group is not a p-group";

Bool,p:=IsPrimePower(Order(G));
K:=Field(M);
error if Characteristic(K) ne p,
  "Error in 'RemoveProjectiveSummands': Characteristic does not match up with group";

KG:=PermutationModule(G,sub<G|>,K);
n:=0;
repeat n+:=1;
  N:=sub<M|Random(M)>;
  if(IsIsomorphic(N,KG)) then
    M:=M/N;
    n:=0;
  end if;
until n eq 20;
return M;
end intrinsic;
```

Save this in a file "pack", say, and then go to a Magma terminal.

```
> Attach("pack");
> RemoveFreeSummands;
Intrinsic 'RemoveFreeSummands'

Signatures:

    (<ModGrp> M) -> ModGrp

        Removes all free summands from a module for a p-group. Returns an error
        if the group is not a p-group and the field's characteristic is not also
        p.



> RemoveFreeSummands(4);

>> RemoveFreeSummands(4);
                     ^
Runtime error in 'RemoveFreeSummands': Bad argument types
Argument types given: RngIntElt

> G:=CyclicGroup(6);
> K:=GF(4);
> RemoveFreeSummands(TrivialModule(G,K));
Error in 'RemoveProjectiveSummands': Group is not a p-group
```

This is still slightly different from the errors that a standard Magma function gives to data that is of the correct type, but not correct. To create those errors, we need the `require` command.

```
intrinsic RemoveFreeSummands(M::ModGrp) -> ModGrp
{Removes all free summands from a module for a p-group. Returns an error if the
group is not a p-group and the field's characteristic is not also p.}
G:=Group(M);
require IsPrimePower(Order(G)): "Group is not a p-group";
Bool,p:=IsPrimePower(Order(G));
K:=Field(M);
require Characteristic(K) eq p: "Characteristic does not match up with group";
KG:=PermutationModule(G,sub<G|>,K);
n:=0;
repeat n+:=1;
```

```
  N:=sub<M|Random(M)>;
   if(IsIsomorphic(N,KG)) then
    M:=M/N;
    n:=0;
  end if;
until n eq 20;
return M;
end intrinsic;
```

```
> Attach("pack");
> G:=CyclicGroup(6); K:=GF(2);
> RemoveFreeSummands(TrivialModule(G,K));

>> RemoveFreeSummands(TrivialModule(G,K));
                     ^
Runtime error in 'RemoveFreeSummands': Group is not a p-group
```

The only thing not covered is how to add parameters to your intrinsics. This would start as so.

```
intrinsic MyFunction(n::RngIntElt, X::SeqEnum[RngIntElt] :
  Method:="Method1", Limit:=10) -> GrpPerm
{Performs some function}
require Type(Method) eq MonStgElt:
  "Parameter Method is not a string";
require Method in {"Method1","Method2"}:
  "Parameter Method must be \"Method1\" or \"Method2\"";
require Type(Limit) eq RngIntElt:
  "Parameter Limit must be an integer";
```

It should be mentioned that packages, once attached, are automatically updated, so there is no need to reattach them if they are updated during the Magma session.