Sage for Abstract Algebra

A Supplement to Abstract Algebra, Theory and Applications

by
Robert A. Beezer
Department of Mathematics and Computer Science
University of Puget Sound

December 23, 2011

Sage Version 4.8
AATA Version 2011-12

Contents

Preface

This supplement explains how to use the open source software Sage to aid in your understanding of abstract algebra. Each section aims to explain Sage commands relevant to some topic in abstract algebra.

This material has been extracted from Tom Judson's open content textbook, *Abstract Algebra: Theory and Applications* and is organized according to the chapters of that text. It also makes frequent reference to examples, definitions and theorems from that text. So this may be useful for learning Sage if you already know some abstract algebra. However, if you are simultaneously learning abstract algebra you will find the textbook useful.

In any event, the best way to use this material is in its electronic form. The content of the text, plus the material here about Sage are available together in an electronic form as a collection of Sage worksheets. These may be uploaded to public Sage servers, such as `sagenb.org`, or used with a Sage installation on your own computer. In this form, the examples printed here are "live" and may be edited and evaluated with new input. Also, you can add your own notes to the text with the built-in word processor available in the Sage notebook.

For more information consult:

- *Abstract Algebra: Theory and Applications*, `http://abstract.pugetsound.edu`
- *Sage*, `http://sagemath.org`

You will also want to be sure you are using a version of Sage that corresponds to the material here. Sage is constantly being improved, and we regularly perform automatic testing of the examples here on the most recent releases. If you are reading the electronic version, you can run the `version()` command below to see which version of Sage you are using (click on the blue "evaluate" link).

```
sage: version()
'Sage Version 4.8.alpha3, Release Date: 2011-12-02'
```

Chapter 1

Preliminaries

1.1 Discussion

Sage is a powerful system for studying and exploring many different areas of mathematics. In this textbook, you will study a variety of algebraic structures, such as groups, rings and fields. Sage does an excellent job of implementing many features of these objects as we will see in the chapters ahead. But here and now, we will concentrate on a few general ways of getting the most out of working with Sage.

1.1.1 Executing Sage Commands

Most of your interaction will be by typing commands into a *compute cell*. That's a compute cell just below this paragraph. Click once inside the compute cell and you will get a more distinctive border around it, a blinking cursor inside, plus a cute little "evaluate" link below it.

At the cursor, type `2+2` and then click on the evaluate link. Did a `4` appear below the cell? If so, you've successfully sent a command off for Sage to evaluate and you've received back the (correct) answer.

Here's another compute cell. Try evaluating the command `factorial(300)` here.

Hmmmmm. That is quite a big integer! The slashes you see at the end of each line mean the result is continued onto the next line, since there are 615 digits in the result.

To make new compute cells, hover your mouse just above another compute cell, or just below some output from a compute cell. When you see a skinny blue bar across the width of your worksheet, click and you will open up a new compute cell, ready for input. Note that your worksheet will remember any calculations you make, in the order you make them, no matter where you put the cells, so it is best to stay organized and add new cells at the bottom.

Try placing your cursor just below the monstrous value of 300! that you have. Click on the blue bar and try another factorial computation in the new compute cell.

Each compute cell will show output due to only the very last command in the cell. Try to predict the following output before evaluating the cell.

```
a = 10
b = 6
b
a = a + 20
a
```

The following compute cell will not print anything since the one command does not create output. But it will have an effect, as you can see when you execute the subsequent cell. Notice how this uses the value of `b` from above. Execute this compute cell *once*. Exactly once. Even if it *appears* to do nothing. If you execute the cell twice, your credit card may be charged twice.

```
b = b + 50
```

Now execute this cell, which will produce some output.

```
b + 20
```

So `b` came into existence as `6`. Then a cell added `50`. This assumes you only executed this cell once! In the last cell we create `b+20` (but do not save it) and it is this value that is output.

You can combine several commands on one line with a semi-colon. This is a great way to get multiple outputs from a compute cell. The syntax for building a matrix should be somewhat obvious when you see the output, but if not, it is not particularly important to understand now.

```
sage: A = matrix([[3, 1], [5,2]]); A

sage: A; print ; A.inverse()
```

1.1.2 Immediate Help

Some commands in Sage are "functions," an example is `factorial()` above. Other commands are "methods" of an object and are like characteristics of objects, an example is `.inverse()` as a method of a matrix. Once you know how to create an object (such as a matrix), then it is easy to see all the available methods. Write the name of the object, place a period ("dot") and hit the TAB key. If you have `A` defined from above, then the compute cell below is ready to go, click into it and then hit TAB (not "evaluate"!). You should get a long list of possible methods.

```
A.
```

To get some help on how to use a method with an object, write its name after a dot (with no parentheses) and then use a question-mark and hit TAB. (Hit the escape key "ESC" to remove the list, or click on the text for a method.)

```
A.inverse?
```

With one more question-mark and a TAB you can see the actual computer instructions that were programmed into Sage to make the method work, once you scoll down past the documentation delimited by the triple quotes (`"""`):

```
A.inverse??
```

It is worthwhile to see what Sage does when there is an error. You will probably see a lot of these at first, and initially they will be a bit intimidating. But with time, you will learn how to use them effectively and you will also become more proficient with Sage and see them less often. Execute the compute cell below, it asks for the inverse of a matrix that has no inverse. Then reread the commentary.

```
sage: B = matrix([[2, 20], [5, 50]])
sage: B.inverse()
```

Click just to the left of the error message to expand it fully (another click hides it totally, and a third click brings back the abbreviated form). Read the bottom of an error message first, it is your best explanation. Here a `ZeroDivisionError` is not 100% accurate, but close. The matrix is not invertible, not dissimilar to how we cannot divide by zero. The remainder of the message begins at the top showing were the error first happened in your code and then the various places where intermediate functions were called, until the actual piece of Sage where the problem occurred. Sometimes this information will give you some clues, sometimes it is totally undecipherable. So do not let it scare you if it seems mysterious.

1.1.3 Annotating Your Work

To comment on your work, you can open up a small word-processor. Hover your mouse until you get the skinny blue bar again, but now when you click, also hold the SHIFT key at the same time. Experiment with fonts, colors, bullet lists, etc and then click the "Save changes" button to exit. Double-click on your text if you need to go back and edit it later.

Open the word-processor again to create a new bit of text (maybe next to the empty compute cell just below). Type all of the following *exactly*, but do not include any backslashes that might precede the dollar signs in the print version:

```
Pythagorean Theorem: \$c^2=a^2+b^2\$
```

and save your changes. The symbols between the dollar signs are written according to the mathematical typesetting language known as TEX — cruise the internet to learn more about this very popular tool. (Well, it is extremely popular among mathematicians and physical scientists.)

1.1.4 Lists

Much of our interaction with sets will be through Sage lists. These are not really sets — they allow duplicates, and order matters. But they are so close to sets, and so easy and powerful to use that we will use them regularly. We will use a fun made-up list for practice, the quote marks mean the items are just text, with no special mathematical meaning. Execute these compute cells as we work through them.

```
sage: zoo = ['snake', 'parrot', 'elephant', 'baboon', 'beetle']
sage: zoo
['snake', 'parrot', 'elephant', 'baboon', 'beetle']
```

So the square brackets define the boundaries of our list, commas separate items, and we can give the list a name. To work with just one element of the list, we use the name and a pair of brackets with an index. Notice that lists have indices that *begin counting at zero*. This will seem odd at first and will seem very natural later.

```
sage: zoo[2]
'elephant'
```

We can add a new creature to the zoo, it is joined up at the far right end.

```
sage: zoo.append('ostrich'); zoo
['snake', 'parrot', 'elephant', 'baboon', 'beetle', 'ostrich']
```

We can remove a creature.

```
sage: zoo.remove('parrot')
sage: zoo
['snake', 'elephant', 'baboon', 'beetle', 'ostrich']
```

We can extract a sublist. Here we start with element 1 (the elephant) and go all the way up to, *but not including*, element 3 (the beetle). Again a bit odd, but it will feel natural later. For now, notice that we are extracting two elements of the lists, exactly $3 - 1 = 2$ elements.

```
sage: mammals = zoo[1:3]
sage: mammals
['elephant', 'baboon']
```

Often we will want to see if two lists are equal. To do that we will need to sort a list first. A function creates a new, sorted list, leaving the original alone. So we need to save the new one with a new name.

```
sage: newzoo = sorted(zoo)
sage: newzoo
['baboon', 'beetle', 'elephant', 'ostrich', 'snake']
```

```
sage: zoo.sort()
sage: zoo
['baboon', 'beetle', 'elephant', 'ostrich', 'snake']
```

Notice that if you run this last compute cell your zoo has changed and some commands above will not necessarily execute the same way. If you want to experiment, go all the way back to the first creation of the zoo and start executing cells again from there with a fresh zoo.

A construction called a "list comprehension" is especially powerful, especially since it almost exactly mirrors notation we use to describe sets. Suppose we want to form the plural of the names of the creatures in our zoo. We build a new list, based on all of the elements of our old list.

```
sage: plurality_zoo = [animal+'s' for animal in zoo]
sage: plurality_zoo
['baboons', 'beetles', 'elephants', 'ostrichs', 'snakes']
```

Almost like it says: we add an "s" to each animal name, for each animal in the zoo, and place them in a new list. Perfect. (Except for getting the plural of "ostrich" wrong.)

1.1.5 Lists of Integers

One final type of list, with numbers this time. The `range()` function will create lists of integers. In its simplest form an invocation like `range(12)` will create a list of 12 integers, *starting at zero* and working up to, *but not including*, 12. Does this sound familiar?

```
sage: dozen = range(12); dozen
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Here are two other forms, that you should be able to understand by studying the examples.

```
sage: teens = range(13, 20); teens
[13, 14, 15, 16, 17, 18, 19]

sage: decades = range(1900, 2000, 10); decades
[1900, 1910, 1920, 1930, 1940, 1950, 1960, 1970, 1980, 1990]
```

1.1.6 Saving and Sharing Your Work

There is a "Save" button in the upper-right corner of your worksheet. This will save a current copy of your worksheet that you can retrieve from within your notebook again later, though you have to re-execute all the cells when you re-open the worksheet later.

There is also a "File" drop-down list, on the left, just above your very top compute cell (not be confused with your browser's File menu item!). You will see a choice here labeled "Save worksheet to a file..." When you do this, you are creating a copy of your worksheet in the "sws" format (short for "Sage WorkSheet"). You can email this file, or post it on a website, for other Sage users and they can use the "Upload"

link on their main notebook page to incorporate a copy of your worksheet into their notebook.

There are other ways to share worksheets that you can experiment with, but this gives you one way to share any worksheet with anybody almost anywhere.

We have covered a lot here in this section, so come back later to pick up tidbits you might have missed. There are also many more features in the notebook that we have not covered.

1.2 Exercises

1 This exercise is just about making sure you know how to use Sage. Login to a notebook server and create a new worksheet. Do some non-trivial computation, maybe a pretty plot or some gruesome numerical computation to an insane precision. Create an interesting list and experiment with it some. Maybe include some nicely formatted text or TEX using the included mini-word-processor (hover until a blue bar appears between cells and then shift-click).

Use whatever mechanism your instructor has in place for submitting your work. Or save your worksheet and then trade worksheets via email (or another electronic method) with a classmate.

Chapter 2

The Integers

2.1 Discussion

Many properties of the algebraic objects we will study can be determined from properties of associated integers. And Sage has many powerful functions for analyzing integers.

2.1.1 Division Algorithm

`a % b` will return the remainder upon division of a by b. In other words, the result is the unique integer r such that (1) $0 \leq r < b$, and (2) $a = bq + r$ for some integer q (the quotient), as guaranteed by the Division Algorithm (Theorem 2.3). Then $(a - r)/b$ will equal q. For example,

```
sage: r = 14 % 3
sage: r
2

sage: q = (14 - r)/3
sage: q
4
```

It is also possible to get both the quotient and remainder at the same time with the `.quo_rem()` method (quotient and remainder).

```
sage: a = 14
sage: b = 3
sage: a.quo_rem(b)
(4, 2)
```

A remainder of zero indicates divisibility. So `(a % b) == 0` will return `True` if b divides a, and will otherwise return `False`.

```
sage: (20 % 5) == 0
True
```

```
sage: (17 % 4) == 0
False
```

The `.divides()` method is another option.

```
sage: c = 5
sage: c.divides(20)
True

sage: d = 4
sage: d.divides(17)
False
```

2.1.2 Greatest Common Divisor

The greatest common divisor of a and b is obtained with the command `gcd(a, b)`, where in our first uses, a and b are integers. Later, a and b can be other objects with a notion of divisibility and "greatness," such as polynomials. For example,

```
sage: gcd(2776, 2452)
4
```

We can use the `gcd` command to determine if a pair of integers are relatively prime.

```
sage: a = 31049
sage: b = 2105
sage: gcd(a, b) == 1
True

sage: a = 3563
sage: b = 2947
sage: gcd(a, b) == 1
False
```

The command `xgcd(a,b)` ("eXtended GCD") returns a triple where the first element is the greatest common divisor of a and b (as with the `gcd(a,b)` command above), but the next two elements are the values of r and s such that $ra + sb = \gcd(a, b)$.

```
sage: xgcd(633,331)
(1, -137, 262)
```

Portions of the triple can be extracted using `[ ]` to access the entries of the triple, starting with the first as number `0`. For example, the following should return the result `True`, even if you change the values of `a` and `b`. Try changing the values of `a` and `b` below, to see that the result is always `True`.

```
sage: a = 633
sage: b = 331
sage: extended = xgcd(a, b)
sage: g = extended[0]
sage: r = extended[1]
sage: s = extended[2]
sage: g == r*a + s*b
True
```

Studying this block of code will go a long way towards helping you get the most out of Sage's output. (Note that `=` is how a value is assigned to a variable, while as in the last line, `==` is how we compare two items for equality.)

2.1.3 Primes and Factoring

The method `.is_prime()` will determine if an integer is prime or not.

```
sage: a = 117371
sage: a.is_prime()
True

sage: b = 14547073
sage: b.is_prime()
False

sage: b == 1597 * 9109
True
```

The command `random_prime(a, proof=True)` will generate a random prime number between 2 and a. Experiment by executing the following two compute cells several times. (Replacing `proof=True` by `proof=False` will speed up the search, but there will be a very, very small probability the result will not be prime.) The `# random` comment is a signal to the automated testing of our examples that the output will be random — you do not need to include that in your own work.

```
sage: a = random_prime(10^21, proof=True)
sage: a                              # random
424729101793542195193

sage: a.is_prime()
True
```

The command `prime_range(a, b)` returns an ordered list of all the primes from a to $b-1$, inclusive. For example,

```
sage: prime_range(500, 550)
[503, 509, 521, 523, 541, 547]
```

The commands `next_prime(a)` and `previous_prime(a)` are other ways to get a single prime number of a desired size. Give them a try in the empty compute cell below.

In addition to checking if integers are prime or not, or generating prime numbers, Sage can also decompose any integer into its prime factors, as described by the Fundamental Theorem of Arithmetic (Theorem 2.8).

```
sage: a = 2600
sage: a.factor()
2^3 * 5^2 * 13
```

So $2600 = 2^3 \times 5^2 \times 13$ and this is the unique way to write 2600 as a product of prime numbers (other than rearranging the order of the primes themselves in the product).

While Sage will print a factorization nicely, it is carried internally as a list of pairs of integers, with each pair being a base (a prime number) and an exponent (a positive integer). Study the following carefully, as it is another good exercise in working with Sage output in the form of lists.

```
sage: a = 2600
sage: factored = a.factor()
sage: first_term = factored[0]
sage: first_term
(2, 3)

sage: second_term = factored[1]
sage: second_term
(5, 2)

sage: third_term = factored[2]
sage: third_term
(13, 1)

sage: first_prime = first_term[0]
sage: first_prime
2

sage: first_exponent = first_term[1]
sage: first_exponent
3
```

The next compute cell reveals the internal version of the factorization by asking for the actual list. And we show how you could determine exactly how many terms the factorization has by using the length command, `len()`.

```
sage: list(factored)
[(2, 3), (5, 2), (13, 1)]

sage: len(factored)
3
```

Can you extract the next two primes and their exponents?

2.2 Exercises

These exercises are about investigating basic properties of the integers, something we will frequently do when investigating groups. Use the editing capabilities of a Sage worksheet to annotate and explain your work.

1 Use the `next_prime()` command to construct two different 8-digit prime numbers.

2 Use the `.is_prime()` method to verify that your primes are really prime.

3 Verify that the greatest common divisor of your two primes is 1.

4 Find two integers that make a "linear combination" of your primes equal to 1. Include a verification of your result.

5 Determine a factorization into powers of primes for $b = 4\,598\,037\,234$.

6 Write statements that show that b is (i) divisible by 7, (ii) not divisible by 11. Your statements should simply return `True` or `False` and be sufficiently general that a different value of b or different candidate prime divisors could be easily used by making just one substitution for each changed value.

Chapter 3

Groups

3.1 Discussion

Many of the groups discussed in this chapter are available for study in Sage. It is important to understand that sets that form algebraic objects (groups in this chapter) are called "parents" in Sage, and elements of these objects are called, well, "elements." So every element belongs to a parent (in other words, is contained in some set). We can ask about properties of parents (finite? order? abelian?), and we can ask about properties of individual elements (identity? inverse?). In the following we will show you how to create some of these common groups and begin to explore their properties with Sage.

3.1.1 Integers mod n

```
sage: Z8 = Integers(8)
sage: Z8
Ring of integers modulo 8

sage: Z8.list()
[0, 1, 2, 3, 4, 5, 6, 7]

sage: a = Z8.an_element(); a
0

sage: a.parent()
Ring of integers modulo 8
```

We would like to work with elements of `Z8`. If you were to type a `6` into a compute cell right now, what would you mean? The integer 6, the rational number $\frac{6}{1}$, the real number 6.00000, or the complex number $6.00000+0.00000i$? Or perhaps you really do want the integer 6 mod 8? Sage really has no idea what you mean or want. To make this clear, you can "coerce" `6` into `Z8` with the syntax `Z8(6)`. Without this, Sage will treat a input number like `6` as an integer, the simplest possible interpretation in some sense. Study the following carefully, where we first work with "normal" integers and then with integers mod 8.

```
sage: a = 6
sage: a
6

sage: a.parent()
Integer Ring

sage: b = 7
sage: c = a + b; c
13

sage: d = Z8(6)
sage: d
6

sage: d.parent()
Ring of integers modulo 8

sage: e = Z8(7)
sage: f = d+e; f
5

sage: g = Z8(85); g
5

sage: f == g
True
```

Z8 is a bit unusual as a first example, since it has two operations defined, both addition and multiplication, with addition forming a group, and multiplication not forming a group. Still, we can work with the additive portion, here forming the Cayley table for the addition.

```
sage: Z8.addition_table(names='elements')
+  0 1 2 3 4 5 6 7
 +---------------
0| 0 1 2 3 4 5 6 7
1| 1 2 3 4 5 6 7 0
2| 2 3 4 5 6 7 0 1
3| 3 4 5 6 7 0 1 2
4| 4 5 6 7 0 1 2 3
5| 5 6 7 0 1 2 3 4
6| 6 7 0 1 2 3 4 5
7| 7 0 1 2 3 4 5 6
```

When n is a prime number, the multipicative structure (excluding zero), will also form a group. The integers mod n are very important, so Sage implements both addition and multiplication together. Groups of symmetries are a better example of how Sage implements groups, since there is just one operation present.

3.1.2 Groups of symmetries

The symmetries of some geometric shapes are already defined in Sage, albeit with different names. They are implemented as "permutation groups" which we will begin to study carefully in Chapter 5.

Sage uses integers to label vertices, starting the count at `1`, instead of letters. Elements by default are printed using "cycle notation" which we will see described carefully in Chapter 5. For now, if `a` is an element of a group of symmetries, then `a.list()` will give the *bottom row* of the notation we are using for permutations. Here is an example, with both the mathematics and Sage. For the Sage part, we create the group of symmetries and then create the symmetry ρ_2 with coercion, followed by outputting the element in cycle notation, and again as a bottom row (coming full circle).

$$\rho_2 = \begin{pmatrix} A & B & C \\ C & A & B \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$$

```
sage: triangle = SymmetricGroup(3)
sage: rho2 = triangle([3,1,2])
sage: rho2
(1,3,2)

sage: rho2.list()
[3, 1, 2]
```

With a list comprehension we can list all six elements of the group in the "bottom row" format. A good exercise would be to pair up each element with its name as given in Figure 3.2.

```
sage: [a.list() for a in triangle]
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Different books, different authors, different software all have different ideas about the order in which to write multiplication of functions. This textbook builds on the idea of composition of functions, so that fg is the composition $(fg)(x) = f(g(x))$ and it is natural to apply g first. Sage takes the opposite view and since we write fg, Sage will understand that we want to do f first. Neither approach is wrong, and neither is necessarily superior, they are just different and there are good arguments for either one. When you consult other books that work with permutation groups, you want to first determine which approach it takes.

The translation here between the text and Sage will be worthwhile practice. Here we will reprise the discussion at the end of Section 3.1, but reverse the order on each product to compute Sage-style and exactly mirror what the text does.

```
sage: mu1 = triangle([1,3,2])
sage: mu2 = triangle([3,2,1])
sage: mu3 = triangle([2,1,3])
```

```
sage: rho1 = triangle([2,3,1])
sage: product = rho1*mu1
sage: product == mu2
True

sage: product.list()
[3, 2, 1]

sage: rho1*mu1 == mu1*rho1
False

sage: mu1*rho1 == mu3
True
```

Now that we understand that Sage does multiplication in reverse, we can compute the Cayley table for this group. Default behavior is to just name elements of a group as letters, `a, b, c`... in the same order that the `.list()` command would produce the elements of the group. But you can also print the elements in the table as themselves (that uses cycle notation here), or you can give the elements names. We will use "u" as shorthand for μ and "r" as shorthand for ρ.

```
sage: triangle.cayley_table()
*  a b c d e f
 +------------
a| a b c d e f
b| b a d c f e
c| c e a f b d
d| d f b e a c
e| e c f a d b
f| f d e b c a

sage: triangle.cayley_table(names='elements')
      *       ()   (2,3)   (1,2) (1,2,3) (1,3,2)   (1,3)
       +-----------------------------------------------
     ()|      ()   (2,3)   (1,2) (1,2,3) (1,3,2)   (1,3)
  (2,3)|   (2,3)      () (1,2,3)   (1,2)   (1,3) (1,3,2)
  (1,2)|   (1,2) (1,3,2)      ()   (1,3)   (2,3) (1,2,3)
(1,2,3)| (1,2,3)   (1,3)   (2,3) (1,3,2)      ()   (1,2)
(1,3,2)| (1,3,2)   (1,2)   (1,3)      () (1,2,3)   (2,3)
  (1,3)|   (1,3) (1,2,3) (1,3,2)   (2,3)   (1,2)      ()

sage: triangle.cayley_table(names=['id','u1','u3','r1','r2','u2'])
 *  id u1 u3 r1 r2 u2
  +-----------------
id| id u1 u3 r1 r2 u2
u1| u1 id r1 u3 u2 r2
```

```
u3| u3 r2 id u2 u1 r1
r1| r1 u2 u1 r2 id u3
r2| r2 u3 u2 id r1 u1
u2| u2 r1 r2 u1 u3 id
```

You should verify that the table above is correct, just like Table 3.2 is correct. Remember that the convention is to multiply a row label times a column label, in that order. However, to do a check across the two tables, you will need to recall the difference in ordering between your textbook and Sage.

3.1.3 Quaternions

Sage implements the quaternions, but the elements are not matrices, but rather are permutations. Despite appearances the structure is identical. It should not matter which version you have in mind (matrices or permutations) if you build the Cayley table and use the default behavior of using letters to name the elements. As permutations, or as letters, can you identify -1, I, J and K?

```
sage: Q = QuaternionGroup()
sage: [x.list() for x in Q]
[[1, 2, 3, 4, 5, 6, 7, 8], [2, 3, 4, 1, 6, 7, 8, 5],
 [3, 4, 1, 2, 7, 8, 5, 6], [4, 1, 2, 3, 8, 5, 6, 7],
 [5, 8, 7, 6, 3, 2, 1, 4], [6, 5, 8, 7, 4, 3, 2, 1],
 [7, 6, 5, 8, 1, 4, 3, 2], [8, 7, 6, 5, 2, 1, 4, 3]]

sage: Q.cayley_table()
*  a b c d e f g h
 +---------------
a| a b c d e f g h
b| b c d a h e f g
c| c d a b g h e f
d| d a b c f g h e
e| e f g h c d a b
f| f g h e b c d a
g| g h e f a b c d
h| h e f g d a b c
```

It should be fairly obvious that `a` is the identity element of the group (1), either from its behavior in the table, or from its "bottom row" representation in the list above. And if you prefer, you can ask Sage.

```
sage: id = Q.identity()
sage: id.list()
[1, 2, 3, 4, 5, 6, 7, 8]
```

Now -1 should have the property that $-1 \cdot -1 = 1$. We see that the identity element `a` is on the diagonal of the Cayley table only when we compute `c*c`. We can verify this easily, borrowing the third "bottom row" element from the list above. With this information, once we locate I, we can easily compute $-I$, and so on.

```
sage: minus_one = Q([3, 4, 1, 2, 7, 8, 5, 6])
sage: minus_one*minus_one == Q.identity()
True
```

See if you can pair up the letters with all eight elements of the quaternions. Be a bit careful with your names, the symbol `I` is used by Sage for the imaginary number i (which we will use below), but Sage will silently let you redefine it to be anything you like. Same goes for lower-case `i`. So call your elements of the quaternions something like `QI, QJ, QK` to avoid confusion.

As we begin to work with groups it is instructive to work with the actual elements. But many properties of groups are totally independent of the order we use for multiplication, or the names or representations we use for the elements. Here are facts about the quaternions we can compute without any knowledge of just how the elements are written or multiplied.

```
sage: Q.is_finite()
True

sage: Q.order()
8

sage: Q.is_abelian()
False
```

3.1.4 Subgroups

The best techniques for creating subgroups will come in future chapters, but we can create some groups that are naturally subgroups of other groups.

Elements of the quaternions were represented by certain permutations of the integers 1 through 8. We can also build the group of *all* permutations of these eight integers. It gets pretty big, so do not list it unless you want a lot of output! (I dare you.)

```
sage: S8 = SymmetricGroup(8)
sage: a = S8.random_element().list()
sage: a                       # random
[5, 2, 6, 4, 1, 8, 3, 7]

sage: S8.order()
40320
```

The quaternions, `Q`, is a subgroup of the full group of all permutations, the symmetric group S_8 or `S8`, and Sage regards this as a property of `Q`.

```
sage: Q.is_subgroup(S8)
True
```

In Sage the complex numbers are known by the name `CC`. We can create a list of the elements in the subgroup described in Example 9. Then we can verify that this set is a subgroup by examining the Cayley table, using multiplication as the operation.

```
sage: H = [CC(1), CC(-1), CC(I), CC(-I)]
sage: CC.multiplication_table(elements=H,
...                           names=['1', '-1', 'i', '-i'])
 *   1 -1  i -i
  +------------
 1|  1 -1  i -i
-1| -1  1 -i  i
 i|  i -i -1  1
-i| -i  i  1 -1
```

3.2 Exercises

These exercises are about becoming comfortable working with groups in Sage.

1 Create the groups `CyclicPermutationGroup(8)` and `DihedralGroup(4)` and give the two groups names of your choosing. We will understand these constructions better shortly, but for now just understand that they are both groups.

2 Check that the groups have the same size with the `.order()` method. Determine which is abelian, and which is not, by using the `.is_abelian()` method.

3 Use the `.cayley_table()` method to create the Cayley table for each group.

4 Write a nicely formatted discussion (Shift-click on a blue bar to bring up the mini-word-processor, use dollar signs to embed bits of TEX) identifying differences between the two groups that are discernible in properties of their Cayley tables. In other words, what is *different* about these two groups that you can "see" in the Cayley tables?

5 For each group, use the `.subgroups()` method to locate a largest subgroup that is not the entire group, and then use the `.list()` method of the subgroup to get a list of elements (which you might save as `elts`).

Now, `.cayley_table(elements=elts)` for the original group will produce the Cayley table of the subgroup. What properties of this table would you check to see if the output is correct?

6 The `.subgroup(elt_list)` method of the original group will create the smallest subgroup containing specified elements of the group, when given the elements as a list `elt_list`. Discover the shortest list of elements necessary to recreate the subgroup you used in the previous exercise. The equality comparison, `==`, can be used to test if two subgroups are equal.

Chapter 4

Cyclic Groups

4.1 Discussion

Cyclic groups are very important, so it is no surprise that they appear in many different forms in Sage. Each is slightly different, and no one implementation is ideal for an introduction, but together they can illustrate most of the important ideas. Here is a guide to the various ways to construct, and study, a cyclic group in Sage.

4.1.1 Infinite Cyclic Groups

In Sage, the integers $\mathbb{Z}$ are constructed with `ZZ`. To build the infinite cyclic group such as $3\mathbb{Z}$ from Example 1, simply use `3*ZZ`. As an infinite set, there is not a whole lot you can do with this. You can test if integers are in this set, or not. You can also recall the generator with the `.gen()` command.

```
sage: G = 3*ZZ
sage: -12 in G
True

sage: 37 in G
False

sage: G.gen()
3
```

4.1.2 Additive Cyclic Groups

The additive cyclic group $\mathbb{Z}_n$ can be built as a special case of a more general Sage construction. First we build $\mathbb{Z}_{14}$ and capture its generator. Throughout, pay close attention to the use of parentheses and square brackets for when you experiment on your own.

```
sage: G = AdditiveAbelianGroup([14])
sage: G.order()
14
```

```
sage: G.list()
[(0), (1), (2), (3), (4), (5), (6), (7),
 (8), (9), (10), (11), (12), (13)]

sage: a = G.gen(0)
sage: a
(1)
```

You can compute in this group, by using the generator, or by using new elements formed by coercing integers into the group, or by taking the result of operations on other elements. And we can compute the order of elements in this group. Notice that we can perform repeated additions with the shortcut of taking integer multiples of an element.

```
sage: a + a
(2)

sage: a + a + a + a
(4)

sage: 4*a
(4)

sage: 37*a
(9)

sage: b = G([2]); b
(2)

sage: b + b
(4)

sage: 2*b == 4*a
True

sage: 7*b
(0)

sage: b.order()
7

sage: c = a - 6*b; c
(3)

sage: c + c + c + c
(12)
```

```
sage: c.order()
14
```

It is possible to create cyclic subgroups, from an element designated to be the new generator. Unfortunately, to do this requires the `.submodule()` method (which should be renamed in Sage).

```
sage: H = G.submodule([b]); H
Additive abelian group isomorphic to Z/7

sage: H.list()
[(0), (2), (4), (6), (8), (10), (12)]

sage: H.order()
7

sage: e = H.gen(0); e
(2)

sage: 3*e
(6)

sage: e.order()
7
```

The cyclic subgroup `H` just created has more than one generator. We can test this by building a new subgroup and comparing the two subgroups.

```
sage: f = 12*a; f
(12)

sage: f.order()
7

sage: K = G.submodule([f]); K
Additive abelian group isomorphic to Z/7

sage: K.order()
7

sage: K.list()
[(0), (2), (4), (6), (8), (10), (12)]

sage: K.gen(0)
(2)

sage: H == K
True
```

Certainly the list of elements, and the common generator of `(2)` lead us to belive that `H` and `K` are the same, but the comparison in the last line leaves no doubt.

Results in this section, especially Theorem 4.6 and Corollary 4.7, can be investigated by creating generators of subgroups from a generator of one additive cyclic group, creating the subgroups, and computing the orders of both elements and orders of groups.

4.1.3 Abstract Multiplicative Cyclic Groups

We can create an abstract cyclic group in the style of Theorems 4.1, 4.2, 4.3. In the syntax below `a` is a name for the generator, and `14` is the order of the element. Notice that the notation is now multiplicative, so we multiply elements, and repeated products can be written as powers.

```
sage: G.<a> = AbelianGroup([14])
sage: G.order()
14

sage: G.list()
[1, a, a^2, a^3, a^4, a^5, a^6, a^7, a^8, a^9, a^10, a^11, a^12, a^13]

sage: a.order()
14
```

Computations in the group are similar to before, only with different notation. Now products, with repeated products written as exponentiation.

```
sage: b = a^2
sage: b.order()
7

sage: b*b*b
a^6

sage: c = a^7
sage: c.order()
2

sage: c^2
1

sage: b*c
a^9

sage: b^37*c^42
a^4
```

Subgroups can be formed with a `.subgroup()` command. But do not try to list the contents of a subgroup, it'll look strangely unfamiliar. Also, comparison of subgroups is not implemented.

```
sage: H = G.subgroup([a^2])
sage: H.order()
7
```

```
sage: K = G.subgroup([a^12])
sage: K.order()
7
```

One advantage of this implementation is the possibility to create all possible subgroups. Here we create the list of subgroups, extract one in particular (the third), and check its order.

```
sage: allsg = G.subgroups(); allsg
[Multiplicative Abelian Group isomorphic to C2 x C7,
 which is the subgroup of Multiplicative Abelian Group
 isomorphic to C14 generated by [a],
 Multiplicative Abelian Group isomorphic to C7,
 which is the subgroup of Multiplicative Abelian Group
 isomorphic to C14 generated by [a^2],
 Multiplicative Abelian Group isomorphic to C2,
 which is the subgroup of Multiplicative Abelian Group
 isomorphic to C14 generated by [a^7],
 Trivial Abelian Group,
 which is the subgroup of Multiplicative Abelian Group
 isomorphic to C14 generated by []]

sage: sub = allsg[2]
sage: sub.order()
2
```

4.1.4 Cyclic Permutation Groups

We will learn more about permutation groups in the next chapter. But we will mention here that it is easy to create cyclic groups as permutation groups, and a variety of methods are available for working with them, even if the actual elements get a bit cumbersome to work with. As before, notice that the notation and syntax is multiplicative.

```
sage: G=CyclicPermutationGroup(14)
sage: a = G.gen(0); a
(1,2,3,4,5,6,7,8,9,10,11,12,13,14)

sage: b = a^2
sage: b = a^2; b
(1,3,5,7,9,11,13)(2,4,6,8,10,12,14)

sage: b.order()
7

sage: a*a*b*b*b
(1,9,3,11,5,13,7)(2,10,4,12,6,14,8)
```

```
sage: c = a^37*b^26; c
(1,6,11,2,7,12,3,8,13,4,9,14,5,10)

sage: c.order()
14
```

We can create subgroups, check their orders, and list their elements.

```
sage: H = G.subgroup([a^2])
sage: H.order()
7

sage: H.gen(0)
(1,3,5,7,9,11,13)(2,4,6,8,10,12,14)

sage: H.list()
[(),
 (1,3,5,7,9,11,13)(2,4,6,8,10,12,14),
 (1,5,9,13,3,7,11)(2,6,10,14,4,8,12),
 (1,7,13,5,11,3,9)(2,8,14,6,12,4,10),
 (1,9,3,11,5,13,7)(2,10,4,12,6,14,8),
 (1,11,7,3,13,9,5)(2,12,8,4,14,10,6),
 (1,13,11,9,7,5,3)(2,14,12,10,8,6,4)]
```

It could help to visualize this group, and the subgroup, as rotations of a regular 12-gon with the vertices labeled with the integers 1 through 12. This is not the full group of symmetries, since it does not include reflections, just the 12 rotations.

4.1.5 Cayley Tables

As groups, each of the examples above (groups and subgroups) should have Cayley tables implemented. Since the groups are cyclic, and their subgroups are therefore cyclic, the Cayley tables should have a similar "cyclic" pattern. Note that the letters used in the default table are generic, and are not related to the letters used above for specific elements — they just match up with the group elements in the order given by `.list()`.

```
sage: G.<a> = AbelianGroup([14])
sage: G.cayley_table()
*  a b c d e f g h i j k l m n
 +----------------------------
a| a b c d e f g h i j k l m n
b| b c d e f g h i j k l m n a
c| c d e f g h i j k l m n a b
d| d e f g h i j k l m n a b c
e| e f g h i j k l m n a b c d
```

```
f| f g h i j k l m n a b c d e
g| g h i j k l m n a b c d e f
h| h i j k l m n a b c d e f g
i| i j k l m n a b c d e f g h
j| j k l m n a b c d e f g h i
k| k l m n a b c d e f g h i j
l| l m n a b c d e f g h i j k
m| m n a b c d e f g h i j k l
n| n a b c d e f g h i j k l m
```

If the real names of the elements are not too complicated, the table could be more informative using these names.

```
sage: K.<b> = AbelianGroup([10])
sage: K.cayley_table(names='elements')
  *    1   b b^2 b^3 b^4 b^5 b^6 b^7 b^8 b^9
   +----------------------------------------
  1|   1   b b^2 b^3 b^4 b^5 b^6 b^7 b^8 b^9
  b|   b b^2 b^3 b^4 b^5 b^6 b^7 b^8 b^9   1
b^2| b^2 b^3 b^4 b^5 b^6 b^7 b^8 b^9   1   b
b^3| b^3 b^4 b^5 b^6 b^7 b^8 b^9   1   b b^2
b^4| b^4 b^5 b^6 b^7 b^8 b^9   1   b b^2 b^3
b^5| b^5 b^6 b^7 b^8 b^9   1   b b^2 b^3 b^4
b^6| b^6 b^7 b^8 b^9   1   b b^2 b^3 b^4 b^5
b^7| b^7 b^8 b^9   1   b b^2 b^3 b^4 b^5 b^6
b^8| b^8 b^9   1   b b^2 b^3 b^4 b^5 b^6 b^7
b^9| b^9   1   b b^2 b^3 b^4 b^5 b^6 b^7 b^8
```

4.1.6 Complex Roots of Unity

The finite cyclic subgroups of $\mathbb{T}$, generated by a primitive nth root of unity are implemented as a more general construction in Sage, known as a cyclotomic field. If you concentrate on just the multiplication of powers of a generator (and ignore the infinitely many other elements) then this is a finite cyclic group. Since this is not implemented directly in Sage as a group, *per se*, it is a bit harder to construct things like subgroups, but it is an excellent exercise to try. It is a nice example since the complex numbers are a concrete and familiar construction. Here are a few sample calculations to provide you with some exploratory tools. See the notes following the computations.

```
sage: G = CyclotomicField(14)
sage: w = G.gen(0); w
zeta14

sage: wc = CDF(w)
sage: wc.abs()
1.0
```

```
sage: wc.arg()/N(2*pi/14)
1.0

sage: b = w^2
sage: b.multiplicative_order()
7

sage: bc = CDF(b); bc
0.623489801859 + 0.781831482468*I

sage: bc.abs()
1.0

sage: bc.arg()/N(2*pi/14)
2.0

sage: sg = [b^i for i in range(7)]; sg
[1, zeta14^2, zeta14^4,
zeta14^5 - zeta14^4 + zeta14^3 - zeta14^2 + zeta14 - 1,
-zeta14, -zeta14^3, -zeta14^5]

sage: c = sg[3]; d = sg[5]
sage: c*d
zeta14^2

sage: c = sg[3]; d = sg[6]
sage: c*d in sg
True

sage: c*d == sg[2]
True

sage: sg[5]*sg[6] == sg[4]
True

sage: G.multiplication_table(elements=sg)
*  a b c d e f g
 +--------------
a| a b c d e f g
b| b c d e f g a
c| c d e f g a b
d| d e f g a b c
e| e f g a b c d
f| f g a b c d e
g| g a b c d e f
```

Notes:

1. `zeta14` is the name of the generator used for the cyclotomic field, it is a primitive root of unity (a 14th root of unity in this case). We have captured it as `w`.

2. The syntax `CDF(w)` will convert the complex number `w` into the more familiar form with real and imaginary parts.

3. The method `.abs()` will return the modulus of a complex number, r as described in the text. For elements of $\mathbb{C}^*$ this should always equal 1.

4. The method `.arg()` will return the argument of a complex number, θ as described in the text. Every element of the cyclic group in this example should have an argument that is an integer multiple of $\frac{2\pi}{14}$. The `N()` syntax converts the symbolic value of `pi` to a numerical approximation.

5. `sg` is a list of elements that form a cyclic subgroup of order 7, composed of the first 7 powers of `b = w^2`. So, for example, the last comparison multiplies the fifth power of `b` with the sixth power of `b`, which would be the eleventh power of `b`. But since `b` has order 7, this reduces to the fourth power. If you know a subset of an infinite group forms a subgroup, then you can produce its Cayley table by specifying the list of elements you want to use. Here we ask for a multiplication table, since that is the relevant operation.

4.2 Exercises

This group of exercises is about the group of units mod n, $U(n)$, which is sometimes cyclic, sometimes not. There are some commands in Sage that will answer some of these questions very quickly, but instead of using those now, just use the basic techniques described. The idea here is to just work with elements, and lists of elements, to discern the subgroup structure of these groups.

1 Execute the statement `U = Integers(40)` to create the set `[0,1,2,...,39]` This is a group under addition mod 40, which we will ignore. Instead we are interested in the subset of elements which have an inverse under *multiplication* mod 40. Determine how big this subgroup is by executing the command `U.unit_group_order()`, and then obtain a list of these elements with `U.list_of_elements_of_multiplicative_group()`.

2 You can create elements of this group by coercing regular integers into `U`, such as with the statement `a = U(7)`. (Don't confuse this with our mathematical notation $U(40)$.) This will tell Sage that you want to view 7 as an element of U, subject to the corresponding operations. Determine the elements of the cyclic subgroup of U generated by 7 with a list comprehension as follows:

```
sage: U = Integers(40)
sage: a = U(7)
sage: [a^i for i in range(16)]
```

What is the order of 7 in $U(40)$?

3 The group $U(49)$ is cyclic. Using only the Sage commands described previously, use Sage to find a generator for this group. Now using *only* theorems about the structure of cyclic groups, describe each of the subgroups of $U(49)$ by specifying its order and by giving an explicit generator. Do not repeat any of the subgroups — in other words, present each subgroup *exactly* once. You can use Sage to check your work on the subgroups, but your answer about the subgroups should rely only on theorems and be a nicely written paragraph with a table, etc.

4 The group $U(35)$ is not cyclic. Again, using only the Sage commands described previously, use computations to provide irrefutable evidence of this. How many of the 16 different subgroups of $U(35)$ can you list?

5 Again, using only the Sage commands described previously, explore the structure of $U(n)$ for various values of n and see if you can formulate an interesting conjecture about some basic property of this group. (Yes, this is a *very* open-ended question, but this is ultimately the real power of exploring mathematics with Sage.)

Chapter 5

Permutation Groups

5.1 Discussion

A good portion of Sage's support for group theory is based on routines from GAP (Groups, Algorithms, and Programming) at http://www.gap-system.org/, which is included in every copy of Sage. This is a mature open source package, dating back to 1986. (Forward reference here to GAP console, etc.)

As we have seen, groups can be described in many different ways, such as sets of matrices, sets of complex numbers, or sets of symbols subject to defining relations. A very concrete way to represent groups is via permutations (one-to-one and onto functions of the integers 1 through n), using function composition as the operation in the group, as described in this chapter. Sage has many routines designed to work with groups of this type and they are also a good way for those learning group theory to gain experience with the basic ideas of group theory. For both these reasons, we will concentrate on these types of groups.

5.1.1 Permutation Groups and Elements

The easiest way to work with permutation group elements in Sage is to write them in cycle notation. Since these are products of disjoint cycles (which commute), we do not need to concern ourselves with the actual order of the cycles. If we write `(1,3)(2,4)` we probably understand it to be a permutation (the topic of this chapter!) and we know that it could be an element of S_4, or perhaps a symmetric group on more symbols than just 4. Sage cannot get started that easily and needs a bit of context, so we coerce a string of characters written with cycle notation into a symmetric group to make group elements. Here are some examples and some sample computations. Remember that Sage and your text differ on the interpretation of the product of two permutations.

```
sage: G = SymmetricGroup(5)
sage: sigma = G("(1,3)(2,5,4)")
sage: sigma*sigma
(2,4,5)
```

```
sage: rho = G("(2,4)(1,5)")
sage: rho^3
(1,5)(2,4)

sage: sigma*rho
(1,3,5,2)

sage: rho*sigma
(1,4,5,3)

sage: rho^-1*sigma*rho
(1,2,4)(3,5)
```

There are alternate ways to create permutation group elements, which can be useful in some situations, but they are not quite as useful in everday use.

```
sage: G = SymmetricGroup(5)
sage: sigma1 = G("(1,3)(2,5,4)")
sage: sigma2 = G([(1,3),(2,5,4)])
sage: sigma3 = G([3,5,1,2,4])
sage: sigma1
(1,3)(2,5,4)

sage: sigma2
(1,3)(2,5,4)

sage: sigma3
(1,3)(2,5,4)

sage: sigma1 == sigma2
True

sage: sigma2 == sigma3
True

sage: sigma2.cycle_tuples()
[(1, 3), (2, 5, 4)]

sage: sigma3.list()
[3, 5, 1, 2, 4]
```

The second version of σ is a list of "tuples", which requires a lot of commas and these must be enclosed in a list. (A tuple of length one must be written like `(4,)` to distinguish it from using parentheses for grouping, as in `5*(4)`.) The third version uses the "bottom-row" of the more cumbersome two-row notation introduced at the beginning of the chapter — it is an ordered list of the *output values* of the permutation when considered as a function.

So we then see that despite three different input procedures, all the versions of σ print the same way, and moreso they are actually equal to each other. (This is a subtle difference — what an object *is* in Sage versus how an object *displays* itself.)

We can be even more careful about the nature of our elements. Notice that once we get Sage started, it can promote the product $\tau\sigma$ into the larger permutation group. We can "promote" elements into larger permutation groups, but it is an error to try to shoe-horn an element into a too-small symmetric group.

```
sage: H = SymmetricGroup(4)
sage: sigma = H("(1,2,3,4)")
sage: G = SymmetricGroup(6)
sage: tau = G("(1,2,3,4,5,6)")
sage: rho = tau * sigma
sage: rho
(1,3)(2,4,5,6)

sage: sigma.parent()
Symmetric group of order 4! as a permutation group

sage: tau.parent()
Symmetric group of order 6! as a permutation group

sage: rho.parent()
Symmetric group of order 6! as a permutation group

sage: tau.parent() == rho.parent()
True

sage: sigmaG = G(sigma)
sage: sigmaG.parent()
Symmetric group of order 6! as a permutation group
```

It is an error to try to coerce a permutation with too many symbols into a permutation group employing too few symbols.

```
sage: tauH = H(tau)
Traceback (most recent call last):
...
ValueError: Invalid permutation vector: (1,2,3,4,5,6)
```

Better than working with just elements of the symmetric group, we can create a variety of permutation groups in Sage. Here is a sampling for starters:

`SymmetricGroup(n)`	Permutations on n symbols, $n!$ elements.
`DihedralGroup(n)`	Symmetries of an n-gon, $2n$ elements.
`CyclicPermutationGroup(n)`	Rotations of an n-gon (no flips), n elements.
`AlternatingGroup(n)`	Alternating group on n symbols, $n!/2$ elements.
`KleinFourGroup()`	A non-cyclic group of order 4.

5.1.2 Properties of Permutation Elements

Sometimes it is easier to grab an element out of a list of elements of a permutation group, and then it is already attached to a parent and there is no need for any coercion. In the following, `rotate` and `flip` are automatically elements of `G` because of the way we procured them.

```
sage: D = DihedralGroup(5)
sage: elements = D.list(); elements
[(), (2,5)(3,4), (1,2)(3,5), (1,2,3,4,5), (1,3)(4,5),
(1,3,5,2,4), (1,4)(2,3), (1,4,2,5,3), (1,5,4,3,2), (1,5)(2,4)]

sage: rotate = elements[3]
sage: flip = elements[1]
sage: flip*rotate == rotate* flip
False
```

So we see from this final statement that the group of symmetries of a pentagon is not abelian. But there is an easier way.

```
sage: D = DihedralGroup(5)
sage: D.is_abelian()
False
```

There are many more methods you can use for both permutation groups and their individual elements. Use the blank compute cell below to create a permutation group (any one you like) and an element of a permutation group (any one you like). Then use tab-completion to see all the methods available for an element, or for a group (name, period, tab-key). Some names you may recognize, some we will learn about in the coming chapters, some are highly-specialized research tools you can use when you write your Ph.D. thesis in group theory. For any of these methods, remember that you can type the name, followed by a question mark, to see documentation and examples. *Experiment and explore* — it is really hard to break anything.

Here are some selected examples of various methods available.

```
sage: A4 = AlternatingGroup(4)
sage: A4.order()
12

sage: A4.is_finite()
True

sage: A4.is_abelian()
False
```

```
sage: A4.is_cyclic()
False

sage: sigma = A4("(1,2,4)")
sage: sigma^-1
(1,4,2)

sage: sigma.order()
3
```

A very useful method when studying the alternating group is the permutation group element method `.sign()`. It will return `1` if a permutation is even and `-1` if a permutation is odd.

```
sage: G = SymmetricGroup(3)
sage: sigma = G("(1,2)")
sage: tau = G("(1,3)")
sage: rho = sigma*tau
sage: sigma.sign()
-1

sage: rho.sign()
1
```

We can create subgroups by giving the main group a list of "generators." These elements serve to "generate" a subgroup — imagine multiplying these elements (and their inverses) together over and over, creating new elements that must also be in the subgroup and also become involved in new products, until you see no new elements. Now that definition ends with a horribly imprecise statement, but it should suffice for now. A better definition is that the subgroup generated by the elements is the smallest subgroup of the main group that contains all the generators — which is fine if you know what all the subgroups might be.

With a single generator, the repeated products just become powers of the lone generator. The subgroup generated then is cyclic. With two (or more) generators, especially in a non-abelian group, the situation can be much, much more complicated. So let's begin with just a single generator. But don't forget to put it in a list anyway.

```
sage: A4 = AlternatingGroup(4)
sage: sigma = A4("(1,2,4)")
sage: sg = A4.subgroup([sigma])
sage: sg
Subgroup of (Alternating group of order 4!/2 as a permutation group)
generated by [(1,2,4)]

sage: sg.order()
3
```

```
sage: sg.list()
[(), (1,2,4), (1,4,2)]

sage: sg.is_abelian()
True

sage: sg.is_cyclic()
True

sage: sg.is_subgroup(A4)
True
```

We can now redo the example from the very beginning of this chapter. We translate to elements to cycle notation, construct the subgroup from two generators (the subgroup is not cyclic), and since the subgroup is abelian, we do not have to view Sage's Cayley table as a diagonal reflection of the table in the example.

```
sage: G = SymmetricGroup(5)
sage: sigma = G("(4,5)")
sage: tau = G("(1,3)")
sage: H = G.subgroup([sigma, tau])
sage: H.list()
[(), (4,5), (1,3), (1,3)(4,5)]

sage: text_names = ['id', 'sigma', 'tau', 'mu']
sage: H.cayley_table(names=text_names)
    *     id sigma   tau    mu
     +------------------------
    id|    id sigma   tau    mu
 sigma| sigma    id    mu   tau
   tau|   tau    mu    id sigma
    mu|    mu   tau sigma    id
```

5.1.3 Motion Group of a Cube

We could mimic the example in the text and create elements of S_4 as permutations of the diagonals. A more obvious, but less insightful, construction is to view the 8 corners of the cube as the items being permuted. Then some obvious symmetries of the cube come from running an axis through the center of a side, through to the center of the obvious side, with quarter-turns or half-turns about these axes forming symmetries. With three such axes and four rotations per axis, we get 12 symmetries, except we have counted the identity permutation two extra times.

Label the four corners of the square top with 1 through 4 with 1 in the left-front corner, arranged clockwise when viewed from above. Use 5 through 8 for the bottom square's corner, so that 5 is below 1, 6 below 2, etc. We will use quarter-turns, clockwise, around each axis, when viewed from above, the front, and the right.

```
sage: G = SymmetricGroup(8)
sage: above = G("(1,2,3,4)(5,6,7,8)")
sage: front = G("(1,4,8,5)(2,3,7,6)")
sage: right = G("(1,2,6,5)(3,7,8,4)")
sage: cube = G.subgroup([above, front, right])
sage: cube.order()
24

sage: cube.list()
[(), (2,4,5)(3,8,6), (2,5,4)(3,6,8), (1,2)(3,5)(4,6)(7,8),
(1,2,3,4)(5,6,7,8), (1,2,6,5)(3,7,8,4), (1,3,6)(4,7,5),
(1,3)(2,4)(5,7)(6,8), (1,3,8)(2,7,5), (1,4,3,2)(5,8,7,6),
(1,4,8,5)(2,3,7,6), (1,4)(2,8)(3,5)(6,7), (1,5,6,2)(3,4,8,7),
(1,5,8,4)(2,6,7,3), (1,5)(2,8)(3,7)(4,6), (1,6,3)(4,5,7),
(1,6)(2,5)(3,8)(4,7), (1,6,8)(2,7,4), (1,7)(2,3)(4,6)(5,8),
(1,7)(2,6)(3,5)(4,8), (1,7)(2,8)(3,4)(5,6), (1,8,6)(2,4,7),
(1,8,3)(2,5,7), (1,8)(2,7)(3,6)(4,5)]
```

Since we know from the discussion in the text that the symmetry group has 24 elements, we see that our three quarter-turns are sufficient to create every symmetry. This prompts several questions:

1. Can you locate the ten rotations about axes? (Hint: the identity is easy, the other 9 never send any symbol to itself.)

2. Can you identify the six symmetries that are a transposition of diagonals? (Hint: `[g for g in cube if g.order() == 2]` is a good preliminary filter.)

3. Verify that any two of our quarter-turns are sufficient to generate the whole group. How do you know they generate the whole group?

4. Can you express one of the diagonal transpositions as a product of quarter-turns? This can be a notoriously difficult problem, especially for software. It is known as the "word problem."

5. Number the six faces of the cube with the numbers 1 through 6 (any way you like). Now consider the same three symmetries we used before (quarter-turns about face-to-face axes) and express them as elements of S_6. Verify that the subgroup generated by these symmetries is the whole group. Again, rather than using three generators, try using just two.

5.2 Exercises

These exercises are designed to help you become familiar with permutation groups in Sage.

1 Create the full symmetric group S_{10} with the command `G = SymmetricGroup(10)`.

2 Create elements of `G` with the following (varying) syntax. Pay attention to commas, quotes, brackets, parentheses. The first two use a string (characters) as input, mimicking the way we write permuations (but with commas). The second two use a list of tuples.

```
a = G("(5,7,2,9,3,1,8)")
b = G("(1,3)(4,5)")
c = G([(1,2),(3,4)])
d = G([(1,3),(2,5,8),(4,6,7,9,10)])
```

(a) Compute a^3, bc, $ad^{-1}b$.
(b) Compute the orders of each of these four individual elements (`a` through `d`) using a single permutation group element method.
(c) Use the permutation group element method `.sign()` to determine if a, b, c, d are even or odd permutations.
(d) Create two cyclic subgroups of G with the commands:

- `H = G.subgroup([a])`
- `K = G.subgroup([d])`

List, and study, the elements of each subgroup. Without using Sage, list the order of each subgroup of K. Then use Sage to construct a subgroup of K with order 10.
(e) More complicated subgroups can be formed by using two or more generators. Construct a subgroup L of G with the command `L = G.subgroup([b,c])`. Compute the order of L and list all of the elements of L.

3 Construct the group of symmetries of the tetrahedron (also the alternating group on 4 symbols, A_4) with the command `A=AlternatingGroup(4)`. Using tools such as orders of elements, and generators of subgroups, see if you can find *all of* the subgroups of A_4 (each one exactly once). Do this without using the `.subgroups()` method to justify the correctness of your answer (though it might be a convenient way to check your work).

Provide a nice summary as your answer - not just piles of output. So use Sage as a tool, as needed, but basically your answer will be a concise paragraph and/or table. This is the one part of this assignment without clear, precise directions, so spend some time on this portion to get it right. Hint: no subgroup of A_4 requires more than two generators.

4 Save your work, and then see if you can crash your Sage session with the commands. Do not submit the list of elements of `N` as part of your submitted worksheet.

- `N = G.subgroup([b,d])`
- `N.list()`

How big is N?

5 Answer the five questions above about the permutations of the cube expressed as permutations of the 8 vertices.

Chapter 6

Cosets and Lagrange's Theorem

6.1 Discussion

Sage can create all of the cosets of a subgroup, and all of the subgroups of a group. While these methods can be somewhat slow, they are in many, many ways much better than experimenting with pencil and paper, and can greatly assist us in understanding the structure of finite groups.

6.1.1 Cosets

Sage will create all the right (or left) cosets of a subgroup. Written mathematically, cosets are sets, and the order of the elements within the set is irrelevant. With Sage, lists are more natural, and here it is to our advantage.

Sage creates the cosets of a subgroup as a list of lists. Each inner list is a single coset. The first coset is always the coset that is the subgroup itself, and the first element of this coset is the identity. Each of the other cosets can be construed to have their first element as their representative, and if you use this element as the representative, the elements of the coset are in the same order they would be created by multiplying this representative by the elements of the first coset (the subgroup).

The keyword `side` can be `'right'` or `'left'`, and if not given, then the default is right cosets. The options refer to which side of the product has the representative. Notice that now Sage's results will be "backwards" compared with the text. Here is Example 2 reprised, but in a slightly different order.

```
sage: G = SymmetricGroup(3)
sage: a = G("(1,2)")
sage: H = G.subgroup([a])
sage: rc = G.cosets(H, side='right'); rc
[[(), (1,2)], [(2,3), (1,3,2)], [(1,2,3), (1,3)]]

sage: lc = G.cosets(H, side='left'); lc
[[(), (1,2)], [(2,3), (1,2,3)], [(1,3,2), (1,3)]]
```

So if we work our way through the brackets carefully we can see the difference between the right cosets and the left cosets. Compare these cosets with the ones in the text and see that left and right are reversed. Shouldn't be a problem — just keep it in mind.

```
sage: G = SymmetricGroup(3)
sage: b = G("(1,2,3)")
sage: H = G.subgroup([b])
sage: rc = G.cosets(H, side='right'); rc
[[(), (1,2,3), (1,3,2)], [(2,3), (1,3), (1,2)]]

sage: lc = G.cosets(H, side='left'); lc
[[(), (1,2,3), (1,3,2)], [(2,3), (1,2), (1,3)]]
```

If we study the backeting, we can see that the left and right cosets are equal. Let's see what Sage thinks:

```
sage: rc == lc
False
```

Mathematically, we need sets, but Sage is working with ordered lists, and the order matters. However, if we know our lists do not have duplicates (the `.cosets()` method will never produce duplicates) then we can sort the lists and a test for equality will perform as expected. The elements of a permutation group have an ordering defined for them — it is not so important *what* this is, just that *some* ordering is defined. The `sorted()` function will take any list and return a sorted version. So for each list of cosets, we will sort the individual cosets and then sort the list of sorted cosets. This is a typical maneuver, though a bit complicated with the nested lists.

```
sage: rc_sorted = sorted([sorted(coset) for coset in rc])
sage: rc_sorted
[[(), (1,2,3), (1,3,2)], [(2,3), (1,2), (1,3)]]

sage: lc_sorted = sorted([sorted(coset) for coset in lc])
sage: lc_sorted
[[(), (1,2,3), (1,3,2)], [(2,3), (1,2), (1,3)]]

sage: rc_sorted == lc_sorted
True
```

The list of all cosets can be quite long (it will include every element of the group) and can take a few seconds to complete, even for small groups. There are more sophisticated, and faster, ways to study cosets (such as just using their representatives), but to understand these techniques you also need to understand more theory.

6.1.2 Subgroups

Sage can compute all of the subgroups of a group. This can produce even more output than the coset method and can sometimes take much longer, depending on the structure of the group. The list is in order of the size of the subgroups, with smallest first. As a demonstration we will first compute and list all of the subgroups of a small group, and then extract just one of these subgroups from the list for some futher study.

```
sage: G = SymmetricGroup(3)
sage: sg = G.subgroups(); sg
[Permutation Group with generators [()],
 Permutation Group with generators [(2,3)],
 Permutation Group with generators [(1,2)],
 Permutation Group with generators [(1,3)],
 Permutation Group with generators [(1,2,3)],
 Permutation Group with generators [(1,2), (1,3,2)]]

sage: H = sg[4]; H
Permutation Group with generators [(1,2,3)]

sage: H.order()
3

sage: H.list()
[(), (1,2,3), (1,3,2)]

sage: H.is_cyclic()
True
```

The output of the `.subgroups()` method can be voluminous, so sometimes we are interested in properties of specific subgroups (as in the previous example) or broader questions of the group's "subgroup structure." Here we expand on Corollary 6.9. Notice that just because Sage does not *compute* a subgroup of order 6 in A_4, this is no substitute whatsoever for a *proof* such as given for the corollary. But the computational result emboldens us to search for the theoretical result with confidence.

```
sage: G = AlternatingGroup(4)
sage: sg = G.subgroups()
sage: [H.order() for H in sg]
[1, 2, 2, 2, 3, 3, 3, 3, 4, 12]
```

So we see no subgroup of oder 6 in the list of subgroups of A_4. Notice how Lagrange's Theorem (Theorem 6.5 is in evidence — all the subgroup orders divide 12, the order of A_4. Be patient, the next subgroup computation may take a while.

```
sage: G = SymmetricGroup(4)
sage: sg = G.subgroups()
sage: [H.order() for H in sg]
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4,
 6, 6, 6, 6, 8, 8, 8, 12, 24]
```

Again, note Lagrange's Theorem in action. But more interestingly, S_4 has a subgroup of order 6. Four of them, to be precise. these four subgroups of order 6 are similar to each other, can you describe them simply (*before* digging into the `sg` list for more information)? If you were curious how many subgroups S_4 has, you could simply count the number of subgroups in the `sg` list. The `len()` function does this for *any* list and is often an easy way to count things.

```
sage: len(sg)
30
```

6.1.3 Subgroups of Cyclic Groups

Now that we are more familiar with permutation groups, and know about the `.subgroups()` method, we can revisit an idea from Chapter 4. The subgroups of a cyclic group are always cyclic, but how many are there and what are their orders?

```
sage: G = CyclicPermutationGroup(20)
sage: [H.order() for H in G.subgroups()]
[1, 2, 4, 5, 10, 20]

sage: G = CyclicPermutationGroup(19)
sage: [H.order() for H in G.subgroups()]
[1, 19]
```

We could do this all day, but you have Sage at your disposal, so vary the order of `G` by changing `n` and study the output across many runs. Maybe try a cyclic group of order 24 and compare with the symmetric group S_4 (above) which also has order 24. Do you feel a conjecture coming on?

```
sage: n = 8
sage: G = CyclicPermutationGroup(n)
sage: [H.order() for H in G.subgroups()]
[1, 2, 4, 8]
```

6.1.4 Euler Phi Function

To add to our number-theoretic functions from Chapter 2, we note that Sage makes the Euler ϕ-function available as the function `euler_phi()`.

```
sage: euler_phi(345)
176
```

Here's an interesting experiment that you can try running several times.

```
sage: m = random_prime(10000)
sage: n = random_prime(10000)
sage: m, n, euler_phi(m*n) == euler_phi(m)*euler_phi(n)  # random
(5881, 1277, True)
```

Feel another conjecture coming on? Can you generalize this result?

6.2 Exercises

The following exercises are less about cosets and subgroups, and more about using Sage as an experimental tool. They are designed to help you become both more efficient, and more expressive, as you write commands in Sage. We will have many opportunities to work with cosets and subgroups in the coming chapters.

These exercises do not contain much guidance, and get more challenging as they go. They are designed to explore, or confirm, results presented in this chapter. You should answer each one with a single (complicated) line of Sage that concludes by outputting `True`.

When you check integers below for divisibility, recognize that `range()` produces plain integers, which are quite simple in their functionality. However, the `srange()` command produces Sage integers, which have many more capabilities. (See the last exercise for an example.)

1 Use `.subgroups()` to find an example of a group G and an integer m, so that (a) m divides the order of G, and (b) G has no subgroup of order m. (Do not use the group A_4 for G, since this is in the text.) Provide a single line of Sage code that has all the logic to produce the desired m as its output. Here is a very simple example that might help you structure your answer.

```
sage: a = 5
sage: b = 10
sage: c = 6
sage: d = 13
sage: a.divides(b)
True

sage: not (b in [c,d])
True

sage: a.divides(b) and not (b in [c,d])
True
```

2 Verify the truth of Fermat's Little Theorem (either variant) for your own choice of a single number for the base a (or b), and for p assuming the value of every prime number between 100 and 1000.

Build up a solution slowly — make a list of powers (start with just a few primes), then make a list of powers reduced by modular arithmetic, then a list of comparisons with the predicted value, then a check on all these logical values resulting from the comparisons. This is a useful strategy for many similar problems. Eventually you will write a single line that performs the verification by eventually printing out `True`. Here are some more hints about useful functions.

```
sage: a = 20
sage: b = 6
sage: a.mod(b)
2

sage: prime_range(50, 100)
[53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

sage: all([True, True, True, True])
True

sage: all([True, True, False, True])
False
```

3 Verify that the group of units mod n has order $n - 1$ when n is prime, again for all primes between 100 and 1000. As before, your output should be simply `True`, just once indicating that the statement about the order is true for all the primes examined. As before, build up your solution slowly, and with a smaller range of primes in the beginning. Express your answer as a single line of Sage code.

4 Verify Euler's Theorem for all values of $0 < n < 100$ and for $1 \leq a \leq n$. This will require nested `for` statements with a conditional. Again, here's a small example that might be helpful for constructing your one-line of Sage code. Note the use of `srange()` in this example.

```
sage: [a/b for a in srange(9) for b in srange(1,a) if gcd(a,b)==1]
[2, 3, 3/2, 4, 4/3, 5, 5/2, 5/3, 5/4, 6, 6/5,
 7, 7/2, 7/3, 7/4, 7/5, 7/6, 8, 8/3, 8/5, 8/7]
```

Chapter 7

Cryptography

7.1 Discussion

Since Sage began as software to support research in number theory, we can quickly and easily demonstrate the internal workings of the RSA algorithm. Recognize that, in practice, many other details such as encoding between letters and integers, or protecting one's private key, are equally important for the security of communications. So RSA itself is just the theoretical foundation.

7.1.1 Constructing Keys

We will suppose that Alice wants to send a secret message to Bob, along with message verification (also known as a message with a digital signature). So we begin with the construction of key pairs (private and public) for both Alice and Bob. We first need two large primes for both individuals, and their product. In practice, values of n would have hundreds of digits, rather than just 21 as we have done here.

```
sage: p_a = next_prime(10^10)
sage: q_a = next_prime(p_a)
sage: p_b = next_prime((3/2)*10^10)
sage: q_b = next_prime(p_b)
sage: n_a = p_a * q_a
sage: n_b = p_b * q_b
sage: n_a, n_b
(100000000520000000627, 225000000300000000091)
```

Computationally, the value of the Euler ϕ-function for a product of primes pq can be obtained from $(p-1)(q-1)$, but we could use Sage's built-in function just as well.

```
sage: m_a = euler_phi(n_a)
sage: m_b = euler_phi(n_b)
sage: m_a, m_b
(100000000500000000576, 225000000270000000072)
```

Now we can create the encryption and decryption exponents. We choose the encryption exponent as a (small) number relatively prime to the value of m. With Sage we can factor m quickly to help us choose this value. In practice we would not want to do this computation for large values of m, so we might more easily choose "random" values and check for the first value which is relatively prime to m. The decryption exponent is the multiplicaive inverse, mod m, of the encryption exponent. If you construct an improper encryption exponent (not relatively prime to m), the computation of the multiplicative inverse will fail (and Sage will tell you so). We do this twice —- for both Alice and Bob.

```
sage: factor(m_a)
2^6 * 3 * 11 * 17 * 131 * 521 * 73259 * 557041

sage: E_a = 5*23
sage: D_a = inverse_mod(E_a, m_a)
sage: D_a
20869565321739130555

sage: factor(m_b)
2^3 * 3^4 * 107 * 1298027 * 2500000001

sage: E_b = 7*29
sage: D_b = inverse_mod(E_b, m_b)
sage: D_b
24384236482463054195
```

At this stage, each individual would publish their values of n and E, while keeping D very private and secure. In practice D might be protected on the user's hard disk (or USB thumb drive they always carry with them) by a password only they know. Every time the person uses D they would need to provide the password. The value of m can be discarded. So for the record, here are all the keys:

```
sage: print "Alice's public key, n:", n_a, " E:", E_a
Alice's public key, n: 100000000520000000627  E: 115

sage: print "Alice's private key, D:", D_a
Alice's private key, D: 20869565321739130555

sage: print "Bob's public key, n:", n_b, " E:", E_b
Bob's public key, n: 225000000300000000091  E: 203

sage: print "Bob's private key, D:", D_b
Bob's private key, D: 24384236482463054195
```

7.1.2 Signing and Encoding a Message

Alice is going to construct a message as an English word with four letters. From these four letters we will construct a single number to represent the message in a form we can use in the RSA algorithm. The function `ord()` will convert a single letter to its ASCII code value, a number between 0 and 127. If we use these numbers as "digits" mod 128, we can be sure that Alice's four-letter word will encode to an integer less than $128^4 = 268,435,456$. The particular maximum value is not important, so long as it is smaller than our value of n since all of our subsequent arithmetic is mod n. We choose a popular four-letter word, convert to ASCII "digits" with a list comprehension, and then construct the integer from the digits with the right base. Notice how we can treat the word as a list and that the first digit in the list is in the "ones" place (we say the list is being treated as in "little-endian" order).

```
sage: word = 'Sage'
sage: digits = [ord(letter) for letter in word]
sage: digits
[83, 97, 103, 101]

sage: message = ZZ(digits, 128)
sage: message
213512403
```

First, Alice will sign her message to provide message verification. She uses her private key for this, since this is an act that only she should be able to perform.

```
sage: signed = power_mod(message, D_a, n_a)
sage: signed
47838774644892618423
```

Then Alice encrypts her message so that only Bob can read it. To do this, she uses Bob's public key. Notice how she does not have to even know Bob — for example, she could have obtained Bob's public key off his web site.

```
sage: encrypted = power_mod(signed, E_b, n_b)
sage: encrypted
111866209291209840488
```

Alice's communication is now ready to travel on any communications network, no matter how insecure it might be.

7.1.3 Decoding and Verifying a Message

Now assume that the value of `encrypted` has reached Bob. Realize that Bob may not know Alice, and realize that Bob does not even necessarily believe what he has received has genuinely originated from Alice. An adversary could be trying to confuse Bob by sending messages that claim to be from Alice. First, Bob must unwrap the

encyption Alice has provided. This is an act only Bob, as the intended recipient, should be able to do. And he does it by using his private key, which only he knows, and which he has kept securely in his possession.

```
sage: decrypted = power_mod(encrypted, D_b, n_b)
sage: decrypted
47838774644892618423
```

Right now, this means very little to Bob. Anybody could have sent him an encoded message. However, this was a message Alice signed. Lets unwrap the message signing. Notice that this uses Alice's public key. Bob does not need to know Alice — for example, he could obtain Alice's key off her web site.

```
sage: received = power_mod(decrypted, E_a, n_a)
sage: received
213512403
```

Bob needs to transform this integer representation back to a word with letters. The `chr()` function converts ASCII code values to letters, and we use a list comprehension to do this repeatedly.

```
sage: digits = received.digits(base=128)
sage: letters = [chr(ascii) for ascii in digits]
sage: letters
['S', 'a', 'g', 'e']
```

If we would like a slightly more recognizable result, we can combine the letters into a string.

```
sage: ''.join(letters)
'Sage'
```

Bob is pleased to obtain such an informative message from Alice. What would have happened if an imposter had sent a message ostensibly from Alice, or what if an adversary had intercepted Alice's original message and replaced it with a tampered message? (The latter is known as a "man in the middle" attack.)

In either case, the rogue party would not be able to duplicate Alice's first action — signing her message. If an adversary somehow signs the message, or tampers with it, the step where Bob unwraps the signing will lead to total garbage. (Try it!) Because Bob received a legitimate word, properly capitalized, he has confidence that the message he unsigned is the same as the message Alice signed. In practice, if Alice sent several hundred words as her message, the odds that it will unsign as cohrent text are astronomically small.

What have we demonstrated?

1. Alice can send messages only Bob can read.

2. Bob can receive secret messages from anybody.

3. Alice can sign messages, so Bob knows they come from Alice.

Of course, without making new keys, you can reverse the roles of Alice and Bob. And if Carol makes a key pair, she can communicate with both Alice and Bob in the same fashion.

If you want to use RSA public-key encryption seriously, investigate the open source software GNU Privacy Guard, aka `GPG`.

7.2 Exercises

1 Construct a keypair for Alice using the first two primes greater than 10^{12}. For your choice of E, use a single prime number and use the smallest possible choice.

Use Sage commands to verify that your encryption and decryption keys are multiplicative inverses.

2 Construct a keypair for Bob using the first two primes greater than $2 \cdot 10^{12}$. For your choice of E, use a single prime number and use the smallest possible choice.

Encode the word `Math` using ASCII values in the same manner as described in this section. Create a signed message of this word for communication from Alice to Bob.

3 Demonstrate how Bob converts the message received from Alice back into the word `Math`.

4 Create a new signed message from Alice to Bob. Simulate the message being tampered with by adding one to the integer Bob receives, before he decrypts it. What result does Bob get for the letters of the message when he decrypts and unsigns the tampered message?

5 Classroom Exercise Organize a class into several small groups. Have each group construct key pairs with some minimum size (digits in n). Each group should keep their private key to themselves, but make their public key available to everybody in the room. It could be written on the board (error-prone) or maybe pasted in a public site like `pastebin.com`. Then each group can send a signed message to another group, where the groups could be arranged logically in a circular fashion for this purpose. Of course, messages should be posted publicly as well. Expect a success rate somewhere between 50% and 100%.

If you do not do this in class, grab a study buddy and send each other messages in the same manner. Expect a success rate of 0%, 50% or 100%.

Chapter 9

Isomorphisms

9.1 Discussion

Sage has limited support for actually creating isomorphisms, though it is possible. However, there is excellent support for determining if two permutation groups are isomorphic. This will allow us to begin a little project to locate *all* of the groups of order less than 16 in Sage s permutation groups.

9.1.1 Isomorphism Testing

If `G` and `H` are two permutation groups, then the command `G.is_isomorphic(H)` will return `True` or `False` as the two groups are, or are not, isomorphic. Since "isomorpic to" is an equivalence relation by Theorem 9.5, it does not matter which group plays the role of `G` and which plays the role of `H`.

So we have a few more examples to work with, let's introduce the Sage command that creates an external direct product. If `G` and `H` are two permutation groups, then the command `direct_product_permgroups([G,H])` will return the external direct product as a new permutation group. Notice that this is a function (not a method) and the input is a list. Rather than just combining two groups in the list, any number of groups can be supplied. We illustrate isomorphism tesing and direct products in the context of Theorem 9.10, which is an equivalence, so tells us *exactly* when we have isomorphic groups. We use cyclic permutation groups as stand-ins for $\mathbb{Z}_n$ by Theorem 9.3.

First, two isomorphic groups.

```
sage: m = 12
sage: n = 7
sage: gcd(m, n)
1

sage: G = CyclicPermutationGroup(m)
sage: H = CyclicPermutationGroup(n)
sage: dp = direct_product_permgroups([G, H])
```

```
sage: K = CyclicPermutationGroup(m*n)
sage: K.is_isomorphic(dp)
True
```

Now, two non-isomorphic groups.

```
sage: m = 15
sage: n = 21
sage: gcd(m, n)
3

sage: G = CyclicPermutationGroup(m)
sage: H = CyclicPermutationGroup(n)
sage: dp = direct_product_permgroups([G, H])
sage: K = CyclicPermutationGroup(m*n)
sage: K.is_isomorphic(dp)
False
```

Notice how the simple computation of a greatest common divisor predicts the incredibly complicated computation of determining if two groups are isomorphic. This is a nice illustration of the power of mathematics, replacing a difficult problem (group isomorphism) by a simple one (factoring and divisibility of integers). Lets build one more direct product of cyclic groups, but with three groups, each with orders that are pairwise relatively prime.

If you try the following with larger parameters you may get an error (`database_gap`).

```
sage: m = 6
sage: n = 5
sage: r = 7
sage: G = CyclicPermutationGroup(m)
sage: H = CyclicPermutationGroup(n)
sage: L = CyclicPermutationGroup(r)
sage: dp = direct_product_permgroups([G, H, L])
sage: K = CyclicPermutationGroup(m*n*r)
sage: K.is_isomorphic(dp)
True
```

9.1.2 Classifying Finite Groups

Once we understand isomorphic groups as being the "same", or "fundamentally no different," or "structurally identical," then it is natural to ask how many "really different" finite groups there are. Corollary 9.4 gives a partial answer: for each prime there is just one finite group, with $\mathbb{Z}_p$ as a concrete manifestation.

Let's embark on a quest to find all the groups of order less than 16 in Sage as permutation groups. For prime orders $1, 2, 3, 5, 7, 11$ and 13 we know there is really just one group each, and we can realize them all:

```
sage: [CyclicPermutationGroup(p) for p in [1, 2, 3, 5, 7, 11, 13]]
[Cyclic group of order 1 as a permutation group,
 Cyclic group of order 2 as a permutation group,
 Cyclic group of order 3 as a permutation group,
 Cyclic group of order 5 as a permutation group,
 Cyclic group of order 7 as a permutation group,
 Cyclic group of order 11 as a permutation group,
 Cyclic group of order 13 as a permutation group]
```

So now our smallest unknown case is order 4. Sage knows at least three such groups, and we can use Sage to check if any pair is isomorphic. Notice that since "isomorphic to" is an equivalence relation, and hence a transitive relation, the two tests below are sufficient.

```
sage: G = CyclicPermutationGroup(4)
sage: H = KleinFourGroup()
sage: T1 = CyclicPermutationGroup(2)
sage: T2 = CyclicPermutationGroup(2)
sage: K = direct_product_permgroups([T1, T2])
sage: G.is_isomorphic(H)
False

sage: H.is_isomorphic(K)
True
```

So we have at least two different groups: $\mathbb{Z}_4$ and $\mathbb{Z}_2 \times \mathbb{Z}_2$, with the latter also known as the Klein 4-group. Sage will not be able to tell us if we have a *complete* list — this will always require theoretical results like Theorem 9.5. We will shortly have a more general result that handles the case of order 4, but right now, a careful analysis (by hand) of the possibilities for the Cayley table of a group of order 4 should lead you to the two possibilities above as the only possibilities. Try to deduce waht the Cayley table of an order 4 group should look like, since you know about identity elements, inverses and cancellation.

We have seen at least two groups of order 6 (next on our list of non-prime orders). One is abelian and one is not, so we do not need Sage to tell us they are structurally different. But let's do it anyway.

```
sage: G = CyclicPermutationGroup(6)
sage: H = SymmetricGroup(3)
sage: G.is_isomorphic(H)
False
```

Is that all? There is $\mathbb{Z}_3 \times \mathbb{Z}_2$, but that is just $\mathbb{Z}_6$ since 2 and 3 are relatively prime. The dihedral group, D_3, all symmetries of a triangle, is just S_3, the symmetric group on 3 symbols.

```
sage: G = DihedralGroup(3)
sage: H = SymmetricGroup(3)
sage: G.is_isomorphic(H)
True
```

It turns out that the two different groups that we know already do form a complete list of all groups of order 6. Again, a future result will make this easy. Also, at order 6 a case-by-case analysis with Cayley tables might test your patience. To Be Continued.

9.1.3 Internal Direct Products

An internal direct product is a statement about subgroups of a single group, together with a theorem that links them to an external direct product. We will work an example here that will illustrate the nature of an internal direct product.

Given an integer n, the set of positive integers less than n, and relatively prime to n forms a group under multiplication mod n. We will work in the set `Integers(n)` where we can add *and* multiply, but we want to stay strictly with multiplication only.

First we build the subgroup itself. Notice how we must convert `x` into an integer (an element of `ZZ`) so that the greatest common divisor computation performs correctly.

```
sage: Z36 = Integers(36)
sage: U = [x for x in Z36 if gcd(ZZ(x), 36) == 1]
sage: U
[1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35]
```

So we have a group of order 12. We are going to try to find a subgroup of order 6 and a subgroup of order 2 to form the internal direct product, and we will restrict our search initially to cyclic subgroups of order 6. Sage has a method that will give the order of each of these elements, relative to multiplication, so let's examine those next.

```
sage: [x.multiplicative_order() for x in U]
[1, 6, 6, 6, 3, 2, 2, 6, 3, 6, 6, 2]
```

We have many choices for generators of a cyclic subgroup of order 6 and for a cyclic subgroup of order 2. Of course, some of the choices for a generator of the subgroup of order 6 will generate the same subgroup. Can you tell, just by counting, how many subgroups of order 6 there are? We are going to pick the first element of order 6, and the last element of order 2, for no particular reason. After your work through this once, we encourage you to try other choices to understand why some choices lead to an internal direct product and some do not. Notice that we choose the elements from the list `U` so that they are sure to be elements of `Z36` and behave properly when multiplied.

```
sage: a = U[1]
sage: b = U[11]
```

```
sage: A = [a^i for i in range(6)]
sage: A
[1, 5, 25, 17, 13, 29]

sage: B = [b^i for i in range(2)]
sage: B
[1, 35]
```

So A and B are two cyclic subgroups. Notice that their intersection is the identity element, one of our requirements for an internal direct product. So this is a good start. Z36 is an abelian group, so A and B are also abelian, thus the condition on all products commuting will hold, but we illustrate the Sage commands that will check this in a non-abelian situation.

```
sage: all([x*y == y*x for x in A for y in B])
True
```

Finally, we need to check that by forming products with elements from A and B we create the entire group. Sorting the resulting list will make a check easier for us visually, and is required if we want Sage to do the check.

```
sage: T = sorted([x*y for x in A for y in B])
sage: T
[1, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35]

sage: T == U
True
```

That's it. We now condense all this information into the statement that "U is the internal direct product of A and B." By Theorem 9.13, we see that U is isomorphic to a product of a cyclic group of order 6 and a cyclic group of oder 2. So in a very real sense, U is no more or less complicated than $\mathbb{Z}_6 \times \mathbb{Z}_2$, which is in turn isomorphic to $\mathbb{Z}_3 \times \mathbb{Z}_2 \times \mathbb{Z}_2$. So we totally understand the "structure" of U. For example, we can see that U is not cyclic, since when written as a product of cyclic groups, the two orders are not relatively prime. The final expression of U suggests you could find three cyclic subgroups of U, with orders 3, 2 and 2, so that U is an internal direct product of the three subgroups.

9.2 Exercises

1 This exercise is about putting Cayley's Theorem into practice. First, read and study the theorem. Realize that this result by itself is primarily of theoretical interest, but with some more theory we could get into some subtler aspects of this (a subject known as "representation theory").

You should create these representations mostly with pencil-and-paper work, using Sage as a fancy calculator and assistant. You do not need to include all these computations in your worksheet. Build the requested group representations and then include enough verifications in Sage to prove that that your representation correctly represents the group.

Begin by building a permutation representation of the quaternions, Q. There are eight elements in Q $(\pm 1, \pm I, \pm J, \pm K)$, so you will be constructing a subgroup of S_8. For each $a \in Q$ form the function λ_a, as defined in the proof of Cayley's theorem. To do this, the two-line version of writing permutations could be useful as an intermediate step. You will probably want to "code" each element of Q with an integer in $\{1, 2, \ldots, 8\}$.

One such representation is included in Sage as `QuaternionGroup()` — your answer should look very similar, but perhaps not identical. Do not submit your answer to this, but I strongly suggest working this particular group representation until you are sure you have it right — the problems below might be very difficult otherwise. You can use Sage's `.is_isomorphic()` method to check if your representations are correct. However, do not use this as a substitute for the part of each question that asks you to investigate properties of your representation towards this end.

(a) Build a permutation representation of $\mathbb{Z}_2 \times \mathbb{Z}_4$ (remember this group is additive, while the theorem uses multiplicative notation). Then construct the group as a subgroup of a full symmetric group generated by exactly two generators. Hint: which two elements of $\mathbb{Z}_2 \times \mathbb{Z}_4$ might you use to generate all of $\mathbb{Z}_2 \times \mathbb{Z}_4$? Use commands in Sage to investigate various properties of your group, other than just `.list()`, to provide evidence that your subgroup is correct — include these in your submitted worksheet.

(b) Build a permutation representation of $U(24)$, the group of units mod 24. Then construct the group as a subgroup of a full symmetric group created with three generators. To determine these three generators, you will likely need to understand $U(24)$ as an internal direct product. Use commands in Sage to investigate various properties of your group, other than just `.list()`, to provide evidence that your subgroup is correct — include these in your submitted worksheet.

2 Consider the symmetries of a 10-gon, D_{10} in your text, `DihedralGroup(10)` in Sage. Identify the permutation that is a 180 degree rotation and use it to generate a subgroup R of order 2. Then identify the permutation that is a 72 degree rotation, and any permutation that is a reflection of the 10-gon about a line. Use these two permutations to generate a subgroup S of order 10. Use Sage to verify that the full dihedral group is the internal direct product of the subgroups R and S.

We have a theorem which says that if a group is an internal direct product, then it is isomorphic to some external direct product. Understand that this does not mean that you can use the converse in this problem. In other words, establishing an isomorphism of G with an external direct product does not prove that G is an internal direct product.

Chapter 10

Normal Subgroups and Factor Groups

10.1 Discussion

Sage has several convenient functions that will allow us to investigate quickly if a subgroup is normal, and if so, the nature of the resulting quotient group. But for an initial understanding, we can also work with the raw cosets. Let's get our hands dirty first, then learn about the easy way.

10.1.1 Multiplying Cosets

The definiton of a factor group requires a normal subgroup, and then we *define* a way to "multiply" two cosets of the subgroup to produce another coset. It is important to realize that we can interpret the definition of a normal subgroup to be *exactly* the condition we need for our new multiplication to be workable. We will do two examples — first with a normal subgroup, then with a subgroup that is not normal.

Consider the dihedral group D_8 that is the symmetry group of an 8-gon. If we take the element that creates a quarter-turn, we can use it generate a cyclic subgroup of order 4. This will be a normal subgroup (trust us for the moment on this). First, build the (right) cosets (notice there is no output):

```
sage: G = DihedralGroup(8)
sage: quarter_turn = G('(1,3,5,7)(2,4,6,8)')
sage: S = G.subgroup([quarter_turn])
sage: C = G.cosets(S)
```

So `C` is a list of lists, with every element of the group `G` occuring exactly once somewhere. You could ask Sage to print out `C` for you if you like, but we will try not to here. We want to multiply two cosets (lists) together. How do we do this? Take *any* element out of the first list, and *any* element out of the second list and multiply them together (which we know how to do since they are elements of `G`). Now we have an element of `G`. What do we do with this element, since we really want a coset as

the result of the product of two cosets? Simple — we see which coset the product is in. Let's give it a try. We will multiply coset 1 with coset 3 (there are 4 cosets by Lagrange's theorem). Study the following code carefully to see if you can understand what it is doing, and *then* read the explanation that follows.

```
sage: p = C[1][0]*C[3][0]
sage: [i for i in range(len(C)) if p in C[i]]
[2]
```

What have we accomplished? In the first line we create `p` as the product of two group elements, one from coset 1 and one from coset 3 (`C[1]`, `C[3]`). Since we can choose *any* element from each coset, we choose the first element of each (`C[ ][0]`). Then we count our way through all the cosets, selecting only cosets that contain `p`. Since `p` will only be in one coset, we expect a list with just one element. Here, our one-element list contains only `2`. So we say the product of coset 1 and coset 3 is coset 2.

The point here is that this result (coset 1 times coset 3 is coset 2) should always be the same, *no matter which elements we pick from the two cosets to form* `p`. So let's do it again, but this time we will not simply choose the first element from each of coset 1 and coset 3, instead we will choose the third element of coset 1 and the second element of coset 3 (we are counting from zero!).

```
sage: p = C[1][2]*C[3][1]
sage: [i for i in range(len(C)) if p in C[i]]
[2]
```

Good. We have the same result. If you are still trusting us on `S` being a normal subgroup of `G`, then this is the result that the theory predicts. Make a copy of the above compute cell and try other choices for the representatives of each coset. Then try the product of other cosets, with varying representatives.

Now is a good time to introduce a way to extend Sage and add new functions. We will design a coset-multiplication function. Read the following carefully and then see the subsequent explanation.

```
sage: def coset_product(i, j, C):
...     p = C[i][0]*C[j][0]
...     c = [k for k in range(len(C)) if p in C[k]]
...     return c[0]
```

The first line creates a new Sage function named `coset_product`. This is accomplished with the word `def`, and note the colon ending the line. The inputs to the function are the numbers of the cosets we want to multiply and the complete list of the cosets. The middle two lines should look familiar from above. We know `c` is a one-element list, so `c[0]` will extract this one coset number, and `return` is what determines that this is the output of the function. Notice that the indentation above

must be exactly as shown. We could have written all this computation on a single line without making a new function, but that begins to get unwieldly. You need to execute the code block above to actually *define* the function, and there will be no output if successful. Now we can use our new function to repeat our work above:

```
sage: coset_product(1, 3, C)
2
```

Now you know the basics of how to add onto Sage and do much more than it was designed for. And with some practice, you could suggest and contribute new functions to Sage, since it is an open source project. Nice.

Now let's examine a situation where the subgroup is not normal. So we will see that our definition of coset multiplication is insufficient in this case. And realize that our new `coset_product` function is also useless since it assumes the cosets come from a normal subgroup.

Consider the alternating group A_4 which we can interpet as the symmetry group of a tetrahedron. For a subgroup, take an element that fixes one vertex and rotates the opposite face — this will generate a cyclic subgroup of order 3, and by Lagrange's Theorem we will get four cosets. Here they are (again, no output is requested here):

```
sage: G = AlternatingGroup(4)
sage: face_turn = G("(1,2,3)")
sage: S = G.subgroup([face_turn])
sage: C = G.cosets(S)
```

Again, let's consider the product of coset 1 and coset 3:

```
sage: p = C[1][0]*C[3][0]
sage: [i for i in range(len(C)) if p in C[i]]
[0]
```

Again, but now for coset 3, choose the second element of the coset to produce the product `p`:

```
sage: p = C[1][0]*C[3][1]
sage: [i for i in range(len(C)) if p in C[i]]
[2]
```

So: is the product of coset 1 and coset 3 equal to coset 0 or coset 2? We cannot say! So there is *no way* to construct a quotient group for this subgroup. You can experiment some more with this subgroup, but in some sense, we are done with this example — there is nothing left to say.

10.1.2 Sage Methods for Normal Subgroups

You can easily ask Sage if a subgroup is normal or not. This is viewed as a property of the subgroup, but you must tell Sage what the "supergroup" is, since the answer can change depending on this value. (For example `H.is_normal(H)` will always be `True`.) Here are our two examples from above.

```
sage: G = DihedralGroup(8)
sage: quarter_turn = G('(1,3,5,7)(2,4,6,8)')
sage: S = G.subgroup([quarter_turn])
sage: S.is_normal(G)
True

sage: G = AlternatingGroup(4)
sage: face_turn = G("(1,2,3)")
sage: S = G.subgroup([face_turn])
sage: S.is_normal(G)
False
```

The text proves in Section 10.2 that A_5 is simple, i.e. A_5 has no normal subgroups. We could build every subgroup of A_5 and ask if it is normal in A_5 using the `.is_normal()` method. But Sage has this covered for us already.

```
sage: G = AlternatingGroup(5)
sage: G.is_simple()
True
```

We can also build a quotient group when we have a normal subgroup.

```
sage: G = DihedralGroup(8)
sage: quarter_turn = G('(1,3,5,7)(2,4,6,8)')
sage: S = G.subgroup([quarter_turn])
sage: Q = G.quotient(S)
sage: Q
Permutation Group with generators [(1,2)(3,4), (1,3)(2,4)]
```

This is useful, but also a bit unsettling. We have the quotient group, but any notion of cosets has been lost, since `Q` is returned as a new permutation group on a different set of symbols. We cannot presume that the numbers used for the new permutation group `Q` bear any resemblance to the cosets we get from the `.cosets()` method. But we can see that the quotient group is described as a group generated by two elements of order two. We could ask for the order of the group, or by Lagrange's Theorem we know the quotient has order 4. We can say now that there are only two groups of order four, the cyclic group of order 4 and a non-cyclic group of order 4, known to us as the Klein 4-group or $\mathbb{Z}_2 \times \mathbb{Z}_2$. This quotient group looks like the non-cyclic one since the cyclic group of order 4 has just one element of order 2. Let's see what Sage says.

```
sage: Q.is_isomorphic(KleinFourGroup())
True
```

Yes, that's it.

Finally, Sage can build us a list of all of the normal subgroups of a group. The list of groups themselves, as we have seen before, is sometimes an overwhelming amount of information. We will demonstrate by just listing the orders of the normal subgroups produced.

```
sage: G = DihedralGroup(8)
sage: N = G.normal_subgroups()
sage: [H.order() for H in N]
[1, 2, 4, 8, 8, 8, 16]
```

So, in particular, we see that our "quarter turn" subgroup is the *only* subgroup of order 4 in this group.

10.2 Exercises

1 Build every subgroup of the alternating group on 5 symbols, A_5, and check that each is not a normal subgroup (except for the two trivial cases). This command could take a while to run — be patient. Compare this with the time needed to run the `.is_simple()` method and realize that there is a significant amount of theory and cleverness brought to bear in speeding up commands like this.

2 Consider the quotient group of the group of symmetries of an 8-gon, formed with the cyclic subgroup of order 4 generated by a quarter-turn. Use the `coset_product` function to determine the Cayley table for this quotient group. Use the number of each coset, as produced by the `.cosets()` method as names for the elements of the quotient group. You will need to build the table "by hand" as there is no easy way to have Sage's Cayley table command do this one for you. You can build a table in the Sage notebook editor (shift-click on a blue line) or you might read the documentation of the `html.table()` method.

3 Consider the cyclic subgroup of order 4 in the symmetries of an 8-gon. Verify that the subgroup is normal by first building the raw left and right cosets (without using the `.cosets()` method) and then checking their equality in Sage, all with a single command that employs sorting with the `sorted()` command.

4 Again, use the same cyclic subgroup of order 4 in the group of symmetries of an 8-gon. Check that the subgroup is normal by using part (3) of Theorem 10.1. Construct a one-line command that does the complete check and returns `True`. Maybe sort the elements of the subgroup `S` first, then slowly build up the necessary lists, commands, and conditions in steps. Notice that this check does not require ever building the cosets.

5 Repeat the demonstration above that for the symmetries of a tetrahedron, the cyclic subgroup of order 3 results in an undefined coset multiplication. Above, the default setting for the `.cosets()` method built right cosets — in this problem, work instead with left cosets. You need to choose two cosets to multiply, and then demonstrate two choices for representatives that lead to different results for the product of the cosets.

6 Construct some dihedral groups of order $2n$ (i.e. symmetries of an n-gon, D_n in the text, `DihedralGroup(n)` in Sage). Maybe for say $2 \leq n \leq 30$. For each dihedral group, construct a list of the orders of each of the normal subgroups (so use `.normal_subgroups()`). Observe enough examples to hypothesize a pattern to your observations, check your hypothesis against each of your examples and then state your hypothesis clearly.

Chapter 11

Homomorphisms

11.1 Discussion

Sage is able to create homomorphisms (and by extension, isomorphisms and automorphisms) between finite permutation groups. There is a limited supply of commands then available to manipulate these functions, but we can still illustrate many of the ideas in this chapter.

11.1.1 Homomorphisms

The principal device for creating a homomorphism is to specify the specific images of the set of generators for the domain. Consider cyclic groups of order 12 and 20:

$$G = \{a^i | a^{12} = e\} \qquad\qquad H = \{x^i | x^{20} = e\}$$

and define a homomorphism by just defining the image of the generator of G, and define the rest of the mapping by extending the mapping via the operation-preserving property of a homomorphism.

$$\begin{aligned} \phi : G \to H, \quad & \phi(a) = x^5 \\ \Rightarrow \quad & \phi(a^i) = \phi(a)^i = (x^5)^i = x^{5i} \end{aligned}$$

The constructor `PermutationGroupMorphism` requires the two groups, then a list of images for each generator (in order!), and then will create the homomorphism. Note that we can then use the result as a function. In the example below, we first verify that `C12` has a single generator (no surprise there), which we then send to a particular element of order 4 in the codomain. Sage then constructs the unique homomorphism that is consistent with this requirement.

```
sage: C12 = CyclicPermutationGroup(12)
sage: C20 = CyclicPermutationGroup(20)
sage: domain_gens = C12.gens()
sage: [g.order() for g in domain_gens]
[12]
```

```
sage: x = C20.gen(0)
sage: y = x^5
sage: y.order()
4

sage: phi = PermutationGroupMorphism(C12, C20, [y])
sage: phi
Permutation group morphism:
  From: Cyclic group of order 12 as a permutation group
  To:   Cyclic group of order 20 as a permutation group
  Defn: [(1,2,3,4,5,6,7,8,9,10,11,12)] ->
        [(1,6,11,16)(2,7,12,17)(3,8,13,18)(4,9,14,19)(5,10,15,20)]

sage: a = C12("(1,6,11,4,9,2,7,12,5,10,3,8)")
sage: phi(a)
(1,6,11,16)(2,7,12,17)(3,8,13,18)(4,9,14,19)(5,10,15,20)

sage: b = C12("(1,3,5,7,9,11)(2,4,6,8,10,12)")
sage: phi(b)
(1,11)(2,12)(3,13)(4,14)(5,15)(6,16)(7,17)(8,18)(9,19)(10,20)

sage: c = C12("(1,9,5)(2,10,6)(3,11,7)(4,12,8)")
sage: phi(c)
()
```

Note that the element `c` must therefore be in the kernel of `phi`.

We can then compute the subgroup of the domain that is the kernel, in this case a cyclic group of order 3 inside the cyclic group of order 12. We can compute the image of *any* subgroup, but here we will build the whole homomorphic image by supplying the whole domain to the `.image()` method. Here the image is a cyclic subgroup of order 4 inside the cyclic group of order 20. Then we can verify the First Isomorphism Theorem.

```
sage: K = phi.kernel(); K
Subgroup of (Cyclic group of order 12 as a permutation group)
generated by [(1,5,9)(2,6,10)(3,7,11)(4,8,12)]

sage: Im = phi.image(C12); Im
Subgroup of (Cyclic group of order 20 as a permutation group)
generated by [(1,6,11,16)(2,7,12,17)(3,8,13,18)(4,9,14,19)(5,10,15,20)]

sage: Im.is_isomorphic(C12.quotient(K))
True
```

Here is a slightly more complicated example. The dihedral group D_{20} is the symmetry group of a 20-gon. Inside this group is a subgroup that is isomorphic to

the symmetry group of a 5-gon (pentagon). Is this a surprise, or is this obvious? Here is a way to make precise the statement "D_{20} contains a copy of D_5."

We build the domain and find its generators, so we know how many images to supply in the definition of the homomorphism. Then we construct the codomain, from which we will construct images. Our choice here is to send a reflection to a reflection, and a rotation to a rotation. But the rotations will both have order 5, and both are a rotation by $\frac{2\pi}{5}$ radians.

```
sage: G = DihedralGroup(5)
sage: H = DihedralGroup(20)
sage: G.gens()
[(1,2,3,4,5), (1,5)(2,4)]

sage: H.gens()
[(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20),
 (1,20)(2,19)(3,18)(4,17)(5,16)(6,15)(7,14)(8,13)(9,12)(10,11)]

sage: x = H.gen(0)^4
sage: y = H.gen(1)
sage: rho = PermutationGroupMorphism(G, H, [x, y])
sage: rho.kernel()
Subgroup of (Dihedral group of order 10 as a permutation group)
generated by [()]
```

Since the kernel is trivial, `rho` is a one-to-one function (see Problem 19). But more importantly, by the First Isomorphishm Theorem, `G` is isomorphic to the image of the homomorphism. We compute the image and check the claim.

```
sage: Im = rho.image(G); Im
Subgroup of (Dihedral group of order 40 as a permutation group)
generated by
[(1,5,9,13,17)(2,6,10,14,18)(3,7,11,15,19)(4,8,12,16,20),
(1,20)(2,19)(3,18)(4,17)(5,16)(6,15)(7,14)(8,13)(9,12)(10,11)]

sage: Im.is_subgroup(H)
True

sage: Im.is_isomorphic(G)
True
```

Just providing a list of images for the generators of the domain is no guarantee that the function will extend to a homomorphism. For starters, the order of each image must divide the order of the corresponding preimage. (Can you prove this?) And similarly, if the domain is abelian, then the image must also be abelian, so in this case the list of images should not generate a non-abelian subgroup. Here is an example. There are no homomorphisms from a cyclic group of order 7 to a cyclic group of order

4 (other than the trivial function that takes every element to the identity). To see this, consider the possible orders of the kernel, and of the two possibilities, see that one is impossible and the other arises with the trivial homomorphism. Unfortunately, Sage acts as if nothing is wrong in creating a homomorphism between these groups, but what Sage builds is useless and raises errors when you try to use it.

```
sage: G = CyclicPermutationGroup(7)
sage: H = CyclicPermutationGroup(4)
sage: tau = PermutationGroupMorphism_im_gens(G, H, H.gens())
sage: tau
Permutation group morphism:
  From: Cyclic group of order 7 as a permutation group
  To:   Cyclic group of order 4 as a permutation group
  Defn: [(1,2,3,4,5,6,7)] -> [(1,2,3,4)]

sage: tau.kernel()
Traceback (most recent call last):
...
RuntimeError: Gap produced error output
...
```

Rather than creating homomorphisms ourselves, in certain situations Sage knows of the existence of natural homomorphisms and will create them for you. One such case is a direct product construction. Given a group `G`, the method `.direct_product(H)` will create the direct product $G \times H$. (This is not the same command as the function `direct_product_permgroups()` from before.) Not only does this command create the direct product, but it also builds *four* homomorphisms, one with domain G, one with domain H and two with domain $G \times H$. So the output consists of five objects, the first being the actual group, and the remainder are homomorphisms. We will demonstrate the call here, and leave a more thorough investigation for the exercises.

```
sage: G = CyclicPermutationGroup(3)
sage: H = DihedralGroup(4)
sage: results = G.direct_product(H)
sage: results[0]
Permutation Group with generators [(4,5,6,7), (4,7)(5,6), (1,2,3)]

sage: results[1]
Permutation group morphism:
  From: Cyclic group of order 3 as a permutation group
  To:   Permutation Group with generators
        [(4,5,6,7), (4,7)(5,6), (1,2,3)]
  Defn: Embedding( Group( [ (1,2,3), (4,5,6,7), (4,7)(5,6) ] ), 1 )

sage: results[2]
Permutation group morphism:
```

```
  From: Dihedral group of order 8 as a permutation group
  To:   Permutation Group with generators
        [(4,5,6,7), (4,7)(5,6), (1,2,3)]
  Defn: Embedding( Group( [ (1,2,3), (4,5,6,7), (4,7)(5,6) ] ), 2 )

sage: results[3]
Permutation group morphism:
  From: Permutation Group with generators
        [(4,5,6,7), (4,7)(5,6), (1,2,3)]
  To:   Cyclic group of order 3 as a permutation group
  Defn: Projection( Group( [ (1,2,3), (4,5,6,7), (4,7)(5,6) ] ), 1 )

sage: results[4]
Permutation group morphism:
  From: Permutation Group with generators
        [(4,5,6,7), (4,7)(5,6), (1,2,3)]
  To:   Dihedral group of order 8 as a permutation group
  Defn: Projection( Group( [ (1,2,3), (4,5,6,7), (4,7)(5,6) ] ), 2 )
```

11.2 Exercises

1 An automorphism is an isomorphism between a group and itself. The identity function ($x \mapsto x$) is always an isomorphism, which we consider trivial. Use Sage to construct a nontrivial automorphism of the cyclic group of order 12. Check that the mapping is both onto and one-to-one by computing the image and kernel and performing the proper tests on these subgroups. Now construct all of the possible automorphisms of the cyclic group of order 12.

2 The four homomorphisms created by the direct product construction are each an example of a more general construction of homomorphisms involving groups G, H and $G \times H$. By using the same groups as in the example above, see if you can discover and describe these constructions with exact definitions of the four homomorphisms in general.

Your tools for investigating a Sage group homomorphism are limited, you might take each generator of the domain and see what its image is. Here is an example of the type of computation you might do repeatedly. We'll investigate the second homomorphism. The domain is the dihedral group, and we will compute the image of the first generator.

```
sage: G = CyclicPermutationGroup(3)
sage: H = DihedralGroup(4)
sage: results = G.direct_product(H)
sage: phi = results[2]
sage: H.gens()
[(1,2,3,4), (1,4)(2,3)]

sage: a = H.gen(0); a
(1,2,3,4)

sage: phi(a)
(4,5,6,7)
```

3 Consider two permutation groups. The first is the subgroup of S_7 generated by $(1, 2, 3)$ and $(4, 5, 6, 7)$. The second is a subgroup of S_{12} generated by $(1, 2, 3)(4, 5, 6)(7, 8, 9)(10, 11, 12)$ and $(1, 10, 7, 4)(2, 11, 8, 5)(3, 12, 9, 6)$. Build these two groups and use the proper Sage command to see that they are isomorphic. Then construct a homomorphism between these two groups that is an isomorphism and include enough details to verify that the mapping is really an isomorphism.

4 The second paragraph of this chapter informally describes a homomorphism from S_n to $\mathbb{Z}_2$, where the even permutations all map to the one of the elements and the odd permutations all map to the other element. Replace S_n by S_6 and replace $\mathbb{Z}_2$ by the permutation version of the cyclic subgroup of order 2, and construct a nontrivial homomorphism between these two groups. Evaluate your homomorphism with enough even and odd permutations to be convinced that it is correct. Then construct the kernel and verify that it is the group you expect.

Hints: First, decide which element of the group of order 2 will be associated with even permutations and which will be associated with odd permutations. Then examine the generators of S_6 to help decide just how to build the homomorphism.

5 The dihedral group D_{20} has several normal subgroups, as seen below. Each of these is the kernel of a homomorphism with D_{20} as the domain. For each normal subgroup of D_{20} construct a homomorphism from D_{20} to D_{20} that has the normal subgroup as the kernel. There is a pattern to many of these, but the three of order 20 will be a challenge.

```
sage: G = DihedralGroup(20)
sage: [H.order() for H in G.normal_subgroups()]
[1, 2, 4, 5, 10, 20, 20, 20, 40]
```

Chapter 13

The Structure of Groups

13.1 Discussion

Cyclic groups, and direct products of cyclic groups, are implemented in Sage as permutation groups. However, these groups quickly become very unwieldly representations and it should be easier to work with finite abelian groups in Sage. So we will postpone any specifics for this chapter until that happens. However, now that we understand the notion of isomorphic groups and the structure of finite abelian groups, we can return to our quest to classify all of the groups with order less than 16.

13.1.1 Classification of Finite Groups

It does not take any sophisticated tools to understand groups of order $2p$, where p is an odd prime. There are two possibilities — a cyclic group of order $2p$ and the dihedral group of order $2p$ that is the set of symmetries of a regular p-gon. The proof requires some close, tight reasoning, but the required theorems are generally just concern orders of elements, Lagrange's Theorem and cosets. This takes care of orders $n = 6$, 10, 14.

For $n = 9$, the upcoming Corollary 14.5 will tell us that any group of order p^2 (where p is a prime) is abelian. So we know from this section that the only two possibilities are $\mathbb{Z}_9$ and $\mathbb{Z}_3 \times \mathbb{Z}_3$. Similarly, the upcoming Theorem 15.8 will tell us that every group of order $n = 15$ is abelian. Now this leaves just one possibility for this order: $\mathbb{Z}_3 \times \mathbb{Z}_5$.

We have just two orders left to analyze: $n = 8$ and $n = 12$. The possibilities are groups we already know, with one exception. However, the analysis that these are the *only* possibilities is more complicated, and will not be pursued now, nor in the next few sections. Notice that $n = 16$ is more complicated still, with 14 different possibilities (which explains why we stopped here).

For $n = 8$ there are 3 abelian groups, and the two non-abelian groups are the dihedral group (symmetries of a square) and the quaternions.

For $n = 12$ there are 2 abelian groups, and 3 non-abelian groups. We know two of the non-abelian groups as a dihedral group, and the alternating group on 4 symbols (which is also the symmetries of a tetrahedron). The third non-abelian group is an

example of a "dicyclic" group, which is an infinite family of groups with order divisible by 4. The order 12 dicyclic group can also be constructed as a "semi-direct product" of two cyclic groups — this is a construction worth knowing as you pursue further study of group theory. The order 8 dicyclic group is also the quaternions and more generally, the dicyclic groups of order 2^k, $k > 2$ are known as "generalized quaternion groups."

The following examples will show you how to construct some of these groups and allows us to make sure the following table is accurate.

```
sage: S = SymmetricGroup(3)
sage: D = DihedralGroup(3)
sage: S.is_isomorphic(D)
True

sage: D1 = CyclicPermutationGroup(3)
sage: D2 = CyclicPermutationGroup(5)
sage: DP = direct_product_permgroups([D1,D2])
sage: C  = CyclicPermutationGroup(15)
sage: DP.is_isomorphic(C)
True

sage: Q  = QuaternionGroup()
sage: DI = DiCyclicGroup(2)
sage: Q.is_isomorphic(DI)
True
```

13.1.2 Groups of Small Order as Permutation Groups

We list here constructions, as permutation groups in Sage, for all of the groups of order less than 16.

n	Construction	Notes, Alternatives
1	`CyclicPermutationGroup(1)`	Trivial
2	`CyclicPermutationGroup(2)`	`SymmetricGroup(2)`
3	`CyclicPermutationGroup(3)`	Prime order
4	`CyclicPermutationGroup(4)`	Cyclic
4	`KleinFourGroup()`	Abelian, non-cyclic
5	`CyclicPermutationGroup(5)`	Prime order
6	`CyclicPermutationGroup(6)`	Cyclic
6	`SymmetricGroup(3)`	Non-abelian `DihedralGroup(3)`
7	`CyclicPermutationGroup(7)`	Prime order
8	`CyclicPermutationGroup(8)`	Cyclic

8	`D1=CyclicPermutationGroup(4)`
	`D2=CyclicPermutationGroup(2)`
	`G=direct_product_permgroups([D1,D2])`
		Abelian, non-cyclic

8	`D1=CyclicPermutationGroup(2)`
	`D2=CyclicPermutationGroup(2)`
	`D3=CyclicPermutationGroup(2)`
	`G=direct_product_permgroups([D1,D2,D3])`
		Abelian, non-cyclic

8	`DihedralGroup(4)`	Non-abelian
8	`QuaternionGroup()`	Quaternions `DiCyclicGroup(2)`
9	`CyclicPermutationGroup(9)`	Cyclic

9	`D1=CyclicPermutationGroup(3)`
	`D2=CyclicPermutationGroup(3)`
	`G=direct_product_permgroups([D1,D2])`
		Abelian, non-cyclic

10	`CyclicPermutationGroup(10)`	Cyclic
10	`DihedralGroup(5)`	Non-abelian
11	`CyclicPermutationGroup(11)`	Prime order
12	`CyclicPermutationGroup(12)`	Cyclic

12	`D1=CyclicPermutationGroup(6)`
	`D2=CyclicPermutationGroup(2)`
	`G=direct_product_permgroups([D1,D2])`
		Abelian, non-cyclic

12	`DihedralGroup(6)`	Non-abelian
12	`AlternatingGroup(4)`	Non-abelian Symmetries of tetrahedron
12	`DiCyclicGroup(3)`	Non-abelian Semi-direct product $Z_3 \rtimes Z_4$
13	`CyclicPermutationGroup(13)`	Prime order
14	`CyclicPermutationGroup(14)`	Cyclic
14	`DihedralGroup(7)`	Non-abelian
15	`CyclicPermutationGroup(15)`	Cyclic

13.2 Exercises

There are no exercises for this section.

Chapter 14

Group Actions

14.1 Discussion

Groups can be realized in many ways, such as as sets of permutations, as sets of matrices, or as sets of abstract symbols related by certain rules ("presentations") and in myriad other ways. We have concentrated on permutation groups because of their concrete feel with elements written as functions and because of their thorough implementation in Sage. Group actions are of great interest when the set they act on is the group itself, and that will be the main application of this chapter in the next chapter. However, any time we have a group action on a set, we can view that group as a permutation group on the elements of the set. So permutation groups are an area of group theory of independent interest with its own definitions and theorems.

We will describe Sage's commands applicable when a group action arises naturally via conjugation, and then move into the more general situation in a more general application.

14.1.1 Conjugation as a Group Action

We might think we need to be careful how Sage defines conjugation (gxg^{-1} versus $g^{-1}xg$) and the difference between Sage and the text on the order of products. However, if you look at the definition of the center and centralizer subgroups you can see that any difference in ordering is irrelevant. Here are the group action commands for the particular action that is conjugation of the elements of the group.

Sage has a permutation group method `.center()` which returns the subgroup of fixed points. The permutation group method, `.centralizer(g)`, returns a subgroup that is the stabilizer of the group element `g`. Finally, the orbits are given by conjugacy classes, but Sage will not flood you with the full conjugacy classes and instead gives back a list of one element per conjugacy class, the representatives, via the permutation group method `.conjugacy_classes_representatives()`. You can manually reconstruct a conjugacy class from a representative, as we do in the example below.

Here is an example of the above commands in action. Notice that an abelian group would be a bad choice for this example.

```
sage: D = DihedralGroup(8)
sage: C = D.center(); C
Subgroup of (Dihedral group of order 16 as a permutation group)
generated by [(1,5)(2,6)(3,7)(4,8)]

sage: C.list()
[(), (1,5)(2,6)(3,7)(4,8)]

sage: a = D("(1,2)(3,8)(4,7)(5,6)")
sage: C1 = D.centralizer(a); C1.list()
[(), (1,2)(3,8)(4,7)(5,6), (1,5)(2,6)(3,7)(4,8), (1,6)(2,5)(3,4)(7,8)]

sage: b = D("(1,2,3,4,5,6,7,8)")
sage: C2 = D.centralizer(b); C2.order()
8

sage: CCR = D.conjugacy_classes_representatives(); CCR
[(), (2,8)(3,7)(4,6), (1,2)(3,8)(4,7)(5,6), (1,2,3,4,5,6,7,8),
 (1,3,5,7)(2,4,6,8), (1,4,7,2,5,8,3,6), (1,5)(2,6)(3,7)(4,8)]

sage: r = CCR[2]; r
(1,2)(3,8)(4,7)(5,6)

sage: conj = []
sage: x = [conj.append(g^-1*r*g) for g in D if not g^-1*r*g in conj]
sage: conj
[(1,2)(3,8)(4,7)(5,6), (1,8)(2,7)(3,6)(4,5), (1,4)(2,3)(5,8)(6,7),
 (1,6)(2,5)(3,4)(7,8)]
```

Notice that in the one conjugacy class constructed all the elements have the same cycle structure, which is no accident. Notice too that `rep` and `a` are the same element, and the product of the order of the centralizer (4) and the size of the conjugacy class (4) equals the order of the group (16), which is a variant of Theorem 14.3.

Verify that the following is a demonstration of the class equation in the special case when the action is conjugation, but would be valid for any group, rather than just `D`.

```
sage: sizes = [D.order()/D.centralizer(g).order()
...                for g in D.conjugacy_classes_representatives()]
sage: sizes
[1, 4, 4, 2, 2, 2, 1]

sage: D.order() == sum(sizes)
True
```

14.1.2 Graph Automorphisms

As mentioned, group actions can be even more interesting when the set they act on is different from the group itself. One class of examples is the group of symmetries of a geometric solid, where the objects in the set are the vertices of the object, or perhaps some other aspect such as edges, faces or diagonals. In this case, the group is all those permutations that move the solid but leave it filling the same space before the motion ("rigid motions").

In this section we will examine something very similar. A **graph** is a mathematical object, consisting of vertices and edges, but the only structure is whether or not any given pair of vertices are joined by an edge or not. The group consists of permutations of vertices that preserve the structure, that is, permutaions of vertices that take edges to edges and non-edges to non-edges. It is very similar to a symmetry group, but there is no notion of any geometric relationships being preserved.

Here is an example. You will need to run the first compute cell to define the graph and get a nice graphic representation.

```
sage: Q = graphs.CubeGraph(3)
sage: Q.plot(layout='spring')      # not tested

sage: A = Q.automorphism_group()
sage: A.order()
48
```

Your plot should look like the vertices and edges of a cube, but may not quite look regular, which is fine, since the geometry is not relevant. Vertices are labeled with strings of three binary digits, 0 or 1, and any two vertices are connected by an edge if their strings differ in exactly one location. We might expect the group of symmetries to have order 24, rather than order 48, given its resemblance to a cube (in appearance and in name). However, when not restricted to rigid motions, we have new permutations that preserve edges. One in particular is to interchange two "opposite faces." Locate two 4-cycles opposite of each other, listed in the same order: $000, 010, 110, 100$ and $001, 011, 111, 101$. Notice that each cycle looks very similar, but all the vertices of the first end in a zero and the second cycle has vertices ending in a one.

Here is how we can see this permutation precisely. You may have noticed that our permutation group has the symbols 1 through 8, while our graph has 8 vertices labeled by binary strings. We build the automorphism group again, but this time we ask for a correspondence ("translation") between the two symbol sets.

```
sage: (A, trans) = Q.automorphism_group(translation=True)
sage: sorted(trans.items())
[('000', 8), ('001', 1), ('010', 2), ('011', 3),
 ('100', 4), ('101', 5), ('110', 6), ('111', 7)]
```

To create the permutation that exchanges the two 4-cycles described above, we use this correspondence to convert from the symbol set for the graph into the symbol

set for the group. Then it is easy to see if the permuation is in the automorphism group.

```
sage: a = PermutationGroupElement("(1,8)(2,3)(4,5)(6,7)")
sage: a in A
True
```

We can use this group to illustrate the relevant Sage commands for group actions.

```
sage: A.orbits()
[[1, 2, 8, 4, 3, 5, 6, 7]]
```

So this action has only one (big) orbit. This implies that every vertex is "like" any other. When a permutation group behaves this way, we say it is "transitive".

```
sage: A.is_transitive()
True
```

If every vertex is "the same" we can compute the stabilizer of any vertex, since they will all be isomorphic. Because vertex 000 is the simplest in some sense, we compute its stabilizer, using symbol 8 in the permutation group.

```
sage: S = A.stabilizer(8)
sage: S.list()
[(), (2,4)(3,5), (1,2)(5,6), (1,2,4)(3,6,5), (1,4,2)(3,5,6),
 (1,4)(3,6)]
```

That `S` has 6 elements is no surprise, since the group has order 48 and the size of the lone orbit is 8. But we can go one step further. The three vertices connected directly to 000 are 100, 010, 001, which are numbered $4, 2, 1$ in the permutation group. Any automorphism of the graph that fixes $000 = 8$ must then permute the three adjacent vertices. There are $3! = 6$ possible ways to do this, and you can check that each appears in one of the six elements of the stabilizer. So we can understand a transitive group by considering the smaller stabilizer, and in this case we can see that each element of the stabilizer is determined by how it permutes the neighbors of the stabilized vertex. Transitive groups are both unusual and important. To contrast, here is a graph automorphism group that is far from transitive (without being trivial). A path is a graph that has all of its vertices in a line. Run the first compute cell to see a path on 11 vertices.

```
sage: P = graphs.PathGraph(11)
sage: P.plot()       # not tested

sage: A, trans = P.automorphism_group(translation=True)
sage: sorted(trans.items())
[(0, 11), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5),
 (6,  6), (7, 7), (8, 8), (9, 9), (10, 10)]
```

```
sage: A.list()
[(), (1,9)(2,8)(3,7)(4,6)(10,11)]
```

The translation is a bit simpler, with the only unusual change being the vertex 0 translated to the permutation group symbol 11. The automorphism group is the trivial identity automorphism (always) and an order 2 permutation that "flips" the path end-to-end. The group is far from transitive and there are many orbits.

```
sage: A.is_transitive()
False

sage: A.orbits()
[[1, 9], [2, 8], [3, 7], [4, 6], [5], [10, 11]]
```

Most of the stabilizers are trivial, with one exception. As subgroups of a group of order 2, there really are not too many options.

```
sage: A.stabilizer(2).list()
[()]

sage: A.stabilizer(5).list()
[(), (1,9)(2,8)(3,7)(4,6)(10,11)]
```

How would this final example have been different if we had used a path on 10 vertices?

14.2 Exercises

1 Construct the Higman-Sims graph with the command `graphs.HigmanSimsGraph()`. Then construct the automorphism group and determine the order of the one interesting normal subgroup of this group. You can try plotting the graph, but the graphic is unlikely to be very informative.

2 This exercise asks you to verify the class equation outside of the usual situation where the group action is conjugation. Consider the example of the automorphism group of the path on 11 vertices. First construct the list of orbits. From each orbit, grab the first element of the orbit as a representative. Compute the size of the orbit as the index of the stabilizer of the representative in the group via Theorem 14.3. (Yes, you could just compute the size of the full orbit, but the idea of the exercise is to use more group-theoretic results.) Then sum these orbit-sizes, which should equal the size of the whole vertex set since the orbits form a partition.

3 Construct a graph, with at least two vertices and at least one edge, whose automorphism group is trivial. You might start by drawing pictures before constructing the graph. A command like the following will let you construct a graph from edges. The graph below looks like a triangle or 3-cycle.

```
sage: G = Graph([(1,2), (2,3), (3,1)])
sage: G.plot()            # not tested
```

4 For the following two pairs of groups, compute the list of conjugacy class representatives for each group in the pair. For each part, compare and contrast the results for the two groups in the pair, with thoughtful and insightful comments.
(a) The full symmetric group on 5 symbols, S_5, and the alternating group on 5 symbols, A_5.
(b) The dihedral groups that are symmetries of a 7-gon and an 8-gon, D_7 and D_8.

5 Use the command `graphs.CubeGraph(4)` to build the four-dimensional cube graph, Q_4. Using a plain `.plot()` command (without a spring layout) should create a nice plot. Construct the automorphism group of the graph and the translation between vertices of the graph and the symbols used in the automorphism group. Then this group (and any of its subgroups) will provide a group action on the vertex set.
(a) Construct the orbits of this action, and comment.
(b) Construct a stabilizer of a single vertex (which is a subgroup of the full automorphism group) and then consider the action of *this* group on the vertex set. Construct the orbits of this new action, and comment carefully and fully on your observations, especially in terms of the vertices of the graph.

6 Build the graph given by the commands below. The result should be a symmetric-looking graph with an automorphism group of order 16.

```
sage: G = graphs.CycleGraph(8)
sage: G.add_edges([(0,2),(1,3),(4,6),(5,7)])
sage: G.plot()                  # not tested
```

Repeat parts (a) and (b) of the previous exercise, but realize that in part (b) there are now two different stabilizers to create, so build both and compare the differences in the stabilizers and their orbits. Creating a second plot with `G.plot(layout='planar')` might provide extra insight.

NOTE: There was a small bug with stabilizers being created as subgroups of symmetric groups on fewer symbols than the correct number. This is fixed in Sage 4.8 and newer. Note the correct output below, and you can check your installation by running the commands.

```
sage: G = SymmetricGroup(4)
sage: S = G.stabilizer(4)
sage: S.orbits()
[[1, 3, 2], [4]]
```

Chapter 15

The Sylow Theorems

15.1 Discussion

15.1.1 Sylow Subgroups

The Sage permutation group method `.sylow_subgroup(p)` will return a single Sylow p-subgroup. If the prime is not a proper divisor of the group order it returns a subgroup of order p^0, in other words, a trivial subgroup. So be careful about the primes you choose. Sometimes, you may only want *one* such Sylow subgroup, since any two Sylow p-subgroups are conjugate, and hence isomorphic (Theorem 15.6). This also means we can create other Sylow p-subgroups by conjugating the one we have. The permutation group method `.conjugate(g)` will conjugate the group by `g`.

With repeated conjugations of a single Sylow p-subgroup, we will likely create duplicate subgroups. So we need to use a slightly complicated construction to form a list of just the unique subgroups in the list of conjugates. The list comprehension will modify the list of unique subgroups, but also create some output we do not care about, so we assign the unwanted output to the variable `junk`.

Lets investigate the Sylow subgroups of the dihedral group D_{18}. As a group of order $36 = 2^2 \cdot 3^2$, we know by the First Sylow Theorem that there is a Sylow 2-subgroup of order 4 and a Sylow 3-subgroup of order 9. First for $p = 2$, we obtain one Sylow 2-subgroup, form all the conjugates, and form a list of non-duplicate subgroups. (These commands take a while to execute, so be patient.)

```
sage: G = DihedralGroup(18)
sage: S2 = G.sylow_subgroup(2); S2
Subgroup of (Dihedral group of order 36 as a permutation group)
generated by
[(2,18)(3,17)(4,16)(5,15)(6,14)(7,13)(8,12)(9,11),
(1,10)(2,11)(3,12)(4,13)(5,14)(6,15)(7,16)(8,17)(9,18)]

sage: allS2 = [S2.conjugate(g) for g in G]
sage: uniqS2 = []
sage: junk = [uniqS2.append(H) for H in allS2 if not H in uniqS2]
sage: uniqS2
```

```
[Permutation Group with generators
[(2,18)(3,17)(4,16)(5,15)(6,14)(7,13)(8,12)(9,11),
(1,10)(2,11)(3,12)(4,13)(5,14)(6,15)(7,16)(8,17)(9,18)],
Permutation Group with generators
[(1,3)(4,18)(5,17)(6,16)(7,15)(8,14)(9,13)(10,12),
(1,10)(2,11)(3,12)(4,13)(5,14)(6,15)(7,16)(8,17)(9,18)],
Permutation Group with generators
[(1,5)(2,4)(6,18)(7,17)(8,16)(9,15)(10,14)(11,13),
(1,10)(2,11)(3,12)(4,13)(5,14)(6,15)(7,16)(8,17)(9,18)],
Permutation Group with generators
[(1,7)(2,6)(3,5)(8,18)(9,17)(10,16)(11,15)(12,14),
(1,10)(2,11)(3,12)(4,13)(5,14)(6,15)(7,16)(8,17)(9,18)],
Permutation Group with generators
[(1,9)(2,8)(3,7)(4,6)(10,18)(11,17)(12,16)(13,15),
(1,10)(2,11)(3,12)(4,13)(5,14)(6,15)(7,16)(8,17)(9,18)],
Permutation Group with generators
[(1,10)(2,11)(3,12)(4,13)(5,14)(6,15)(7,16)(8,17)(9,18),
(1,11)(2,10)(3,9)(4,8)(5,7)(12,18)(13,17)(14,16)],
Permutation Group with generators
[(1,10)(2,11)(3,12)(4,13)(5,14)(6,15)(7,16)(8,17)(9,18),
(1,13)(2,12)(3,11)(4,10)(5,9)(6,8)(14,18)(15,17)],
Permutation Group with generators
[(1,10)(2,11)(3,12)(4,13)(5,14)(6,15)(7,16)(8,17)(9,18),
(1,15)(2,14)(3,13)(4,12)(5,11)(6,10)(7,9)(16,18)],
Permutation Group with generators
[(1,10)(2,11)(3,12)(4,13)(5,14)(6,15)(7,16)(8,17)(9,18),
(1,17)(2,16)(3,15)(4,14)(5,13)(6,12)(7,11)(8,10)]]

sage: len(uniqS2)
9
```

The Third Sylow Theorem tells us that for $p = 2$ we would expect $1, 3$ or 9 Sylow 2-subgroups, so our computational result of 9 subgroups is consistent with what the theory predicts. Can you visualize each of these subgroups as symmetries of an 18-gon? Notice that we also have many subgroups of order 2 inside of these subgroups of order 4.

```
sage: G = DihedralGroup(18)
sage: S3 = G.sylow_subgroup(3); S3
Subgroup of (Dihedral group of order 36 as a permutation group)
generated by
[(1,7,13)(2,8,14)(3,9,15)(4,10,16)(5,11,17)(6,12,18),
(1,15,11,7,3,17,13,9,5)(2,16,12,8,4,18,14,10,6)]

sage: allS3 = [S3.conjugate(g) for g in G]
sage: uniqS3 = []
```

```
sage: junk = [uniqS3.append(H) for H in allS3 if not H in uniqS3]
sage: uniqS3
[Permutation Group with generators
[(1,7,13)(2,8,14)(3,9,15)(4,10,16)(5,11,17)(6,12,18),
(1,15,11,7,3,17,13,9,5)(2,16,12,8,4,18,14,10,6)]]

sage: len(uniqS3)
1
```

What does the Third Sylow Theorem predict? Just 1 or 4 Sylow 3-subgroups. Having found just one subgroup computationally, we know that all of the conjugates of the lone Sylow 3-subgroup are equal. In other words, the Sylow 3-subgroup is normal in D_{18}. Let's check.

```
sage: S3.is_normal(G)
True
```

At least one of the subgroups of order 3 contained in this Sylow 3-subgroup should be obvious by looking at the orders of the generators, and then you may even notice that the generators given could be reduced, and one is a power of the other.

```
sage: S3.is_cyclic()
True
```

Remember that there are many other subgroups, of other orders. For example, can you construct a subgroup of order $6 = 2 \cdot 3$ in D_{18}?

15.1.2 Normalizers

A new command that is relevant to this section is the construction of a normalizer. The Sage command `G.normalizer(H)` will return the subgroup of `G` containing elements that normalize the subgroup `H`. We illustrate its use with the Sylow subgroups from above.

```
sage: G = DihedralGroup(18)
sage: S2 = G.sylow_subgroup(2)
sage: S3 = G.sylow_subgroup(3)
sage: N2 = G.normalizer(S2); N2
Subgroup of (Dihedral group of order 36 as a permutation group)
generated by
[(2,18)(3,17)(4,16)(5,15)(6,14)(7,13)(8,12)(9,11),
(1,10)(2,11)(3,12)(4,13)(5,14)(6,15)(7,16)(8,17)(9,18)]

sage: N2 == S2
True
```

```
sage: N3 = G.normalizer(S3); N3
Subgroup of (Dihedral group of order 36 as a permutation group)
generated by
[(2,18)(3,17)(4,16)(5,15)(6,14)(7,13)(8,12)(9,11),
(1,2)(3,18)(4,17)(5,16)(6,15)(7,14)(8,13)(9,12)(10,11),
(1,7,13)(2,8,14)(3,9,15)(4,10,16)(5,11,17)(6,12,18),
(1,15,11,7,3,17,13,9,5)(2,16,12,8,4,18,14,10,6)]

sage: N3 == G
True
```

The normalizer of a subgroup always contains the whole subgroup, so the normalizer of `S2` is as small as possible. We already knew `S3` is normal in `G`, so it is no surprise that its normalizer is as big as possible — every element of `G` normalizes `S3`. Let's compute a normalizer in D_{18} that is more "interesting."

```
sage: G = DihedralGroup(18)
sage: a = G("(1,7,13)(2,8,14)(3,9,15)(4,10,16)(5,11,17)(6,12,18)")
sage: b = G("(1,5)(2,4)(6,18)(7,17)(8,16)(9,15)(10,14)(11,13)")
sage: H = G.subgroup([a, b])
sage: H.order()
6

sage: N = G.normalizer(H)
sage: N
Subgroup of (Dihedral group of order 36 as a permutation group)
generated by
[(1,2)(3,18)(4,17)(5,16)(6,15)(7,14)(8,13)(9,12)(10,11),
(1,5)(2,4)(6,18)(7,17)(8,16)(9,15)(10,14)(11,13),
(1,13,7)(2,14,8)(3,15,9)(4,16,10)(5,17,11)(6,18,12)]

sage: N.order()
12
```

So for this subgroup of order 6, the normalizer is strictly bigger than the subgroup, but still strictly smaller than the whole group (and hence not normal in the dihedral group). Trivially, a subgroup is normal in its normalizer:

```
sage: H.is_normal(G)
False

sage: H.is_normal(N)
True
```

15.1.3 Finite Simple Groups

We saw earlier Sage's permutation group method `.is_simple()`. Example 7 tells us that a group of order 64 is never simple. The dicyclic group `DiCyclicGroup(16)` is a non-abelian group of 64, so we can test this method on this group. It turns out this group has many normal subgroups — the list will always contain the trivial subgroup and the group itself, so any number exceeding 2 indicates a non-trivial normal subgroup.

```
sage: DC=DiCyclicGroup(16)
sage: DC.order()
64

sage: DC.is_simple()
False

sage: ns = DC.normal_subgroups()
sage: len(ns)
9
```

Here is a rather interesting group, one of the 26 sporadic simple groups, known as the Higman-Sims group, HS. The generators used below come from the representation on 100 points in GAP format, available off of http://web.mat.bham.ac.uk/atlas/v2.0/spor/HS/. Generators of order 2 and order 5, roughly 44 million elements, but no normal subgroups. Amazing.

```
sage: G = SymmetricGroup(100)
sage: a = G([(1,60),  (2,72),  (3,81),  (4,43),  (5,11),  (6,87),
...          (7,34),  (9,63),  (12,46), (13,28), (14,71), (15,42),
...          (16,97), (18,57), (19,52), (21,32), (23,47), (24,54),
...          (25,83), (26,78), (29,89), (30,39), (33,61), (35,56),
...          (37,67), (44,76), (45,88), (48,59), (49,86), (50,74),
...          (51,66), (53,99), (55,75), (62,73), (65,79), (68,82),
...          (77,92), (84,90), (85,98), (94,100)])
sage: b = G([(1,86,13,10,47),  (2,53,30,8,38),
...          (3,40,48,25,17),  (4,29,92,88,43),   (5,98,66,54, 65),
...          (6,27,51,73,24),  (7,83,16,20,28),   (9,23,89,95,61),
...          (11,42,46,91,32), (12,14, 81,55,68), (15,90,31,56,37),
...          (18,69,45,84,76), (19,59,79,35,93),  (21,22,64,39,100),
...          (26,58,96,85,77), (33,52,94,75,44),  (34,62,87,78,50),
...          (36,82,60,74,72), (41,80,70,49,67),  (57,63,71,99,97)])
sage: a.order()
2

sage: b.order()
5
```

```
sage: HS = G.subgroup([a, b])
sage: HS.order()
44352000

sage: HS.is_simple()
True
```

We saw this group earlier in the exercises for Chapter 14 on group actions, where it was the single non-trivial normal subgroup of the automorphism group of the Higman-Sims graph.

15.1.4 GAP Console and Interface

This concludes our exclusive study of group theory, though we will be using groups some in the subsequent sections. As we have remarked, much of Sage's computation with groups is performed by the open source program, "Groups, Algorithms, and Programming," which is better know as simply GAP. If after this course you outgrow Sage's support for groups, then learning GAP would be your next step as a group theorist. Every copy of Sage includes a copy of GAP and is easy to see which version of GAP is included:

```
sage: gap.version()
'4.4.12'
```

You can interact with GAP in Sage in several ways. The most direct is by creating a permutation group via Sage's `gap()` command.

```
sage: G = gap('Group( (1,2,3,4,5,6), (1,3,5) )')
sage: G
Group( [ (1,2,3,4,5,6), (1,3,5) ] )
```

Now we can use most any GAP command with `G`, via the convention that most GAP commands expect a group as the first argument, and we instead provide the group by using the `G.` syntax. If you consult the GAP documentation you will see that `Center` is a GAP command that expects a group as its lone argument, and `Centralizer` is a GAP command that expects two arguments — a group and then a group element.

```
sage: G.Center()
Group( [ ( 1, 3, 5)( 2, 4, 6) ] )

sage: G.Centralizer('(1, 3, 5)')
Group( [ (1,3,5), (2,4,6), (1,3,5)(2,4,6) ] )
```

In a worksheet you can set the first line of a compute cell to `%gap` and the entire cell will be interpreted as if you were interacting directly with GAP. This means you would now have to use GAP's syntax. You can also use the drop-down box at the top of a worksheet, and select `gap` as the system (rather than `sage`) and the whole worksheet will be interpreted as GAP commands. Here is one simple example, which you should be able to evaluate in your notebook.

```
%gap
G := Group( (1,2,3,4,5,6), (1,3,5) );
Centralizer(G, (1,3,5));
```

Notice that

1. We do not need to wrap the individual permutations in as many quotation marks as we do in Sage.

2. Assignment is `:=` not `=`. If you forget the colon, you will get an error message such as `Variable: 'G' must have a value`.

3. A line *must* end with a semi-colon. If you forget, several lines will be merged together.

You can get help about GAP commands with a command such as the following, though you will soon see that GAP assumes you know a lot more algebra than Sage assumes you know.

```
print gap.help('SymmetricGroup', pager=False)
```

In the command-line version of Sage, you can also use the GAP "console." Again, you need to use GAP syntax, and you do not have many of the conveniences of the Sage notebook. It is also good to know in advance that `quit;` is how you can leave the GAP console and get back to Sage. If you run Sage at the command-line, use the command `gap_console()` to start GAP running.

It is a comfort to know that with Sage you get a complete copy of GAP, installed and all ready to run. However, this is not a tutorial on GAP, so consult the documentation available at the main GAP website: www.gap-system.org to learn how to get the most out of it.

15.2 Exercises

1 This exercise verifies Theorem 15.9. The commutator subgroup is computed with the permutation group method `.commutator()`. For the dihedral group of order 40, D_{20} (`DihedralGroup(20)` in Sage), compute the commutator subgroup and form the quotient with the dihedral group. Then verify that this quotient is abelian. Can you identify the quotient group exactly (in other words, up to isomorphism)?

2 For each possible prime, find all of the distinct Sylow p-subgroups of the alternating group A_5. Confirm that your results are consistent with the Third Sylow Theorem for each prime. We know that A_5 is a simple group. Explain how this would explain or predict some aspects of your answers.

Count the number of distinct elements contained in the union of all the Sylow subgroups you just found. What is interesting about this count?

3 For each possible prime, find all of the distinct Sylow p-subgroups of the dihedral group D_{36} (symmetries of a 36-gon) for each possible prime. Confirm that your results are consistent with the Third Sylow Theorem for each prime. It can be proved that *any group* with order 72 is not a simple group, using techniques such as those used in the later examples in this chapter. Explain how this result would explain or predict some aspects of your answers.

4 This exercise verifies Lemma 15.5. Let G be the dihedral group of order 36, D_{18}. Let H be the one Sylow 3-subgroup. Let K be the subgroup of order 6 generated by the two permutations `a` and `b` given below. First, form a list of the distinct conjugates of K by the elements of H, and determine the number of subgroups in this list. Compare this with the index given in the statement of the lemma, employing a single (long) statement making use of the `.order()`, `.normalizer()` and `.intersection()` methods.

```
sage: G = DihedralGroup(18)
sage: a = G("(1,7,13)(2,8,14)(3,9,15)(4,10,16)(5,11,17)(6,12,18)")
sage: b = G("(1,5)(2,4)(6,18)(7,17)(8,16)(9,15)(10,14)(11,13)")
```

5 Example 9 shows that every group of order 48 has a normal subgroup. The dicyclic groups are an infinite family of non-abelian groups with order $4n$, which includes the quaternions when $n = 2$. So the permutation group `DiCyclicGroup(12)` has order 48. Use Sage to follow the logic of the proof in Example 9 and construct a normal subgroup in this group. (In other words, do not just ask for a list of the normal subgroups, but trace through the implications in the example to arrive at the normal subgroup, and check your answer.)

Chapter 16

Rings

16.1 Discussion

Rings are very important in your study of abstract algebra, and similarly, they are very important in the design and use of Sage. There is a lot of material in this chapter, and there are many corresponding commands in Sage.

16.1.1 Creating Rings

Here is a list of various rings, domains and fields you can construct simply.

1. `Integers()`, `ZZ`: the integral domain of positive and negative integers, $\mathbb{Z}$.

2. `Integers(n)`: the integers mod n, $\mathbb{Z}_n$. A field when n is prime, but just a ring for composite n.

3. `QQ`: the field of rational numbers, $\mathbb{Q}$.

4. `RR`, `CC`: the field of real numbers and the field of complex numbers, $\mathbb{R}$, $\mathbb{C}$. It is impossible to create *every* real number inside a computer, so technically these sets do not behave as fields, but only give a good imitiation of the real thing. We say they are "inexact" rings to make this point.

5. `QuadraticField(n)`: the field formed by combining the rationals with a solution to the polynomial equation $x^2 - n = 0$. The notation in the text is $\mathbb{Q}[\sqrt{n}]$. A functional equivalent can be made with the syntax `QQ[sqrt(n)]`. Note that `n` can be negative.

6. `CyclotomicField(n)`: the field formed by combining the rationals with the solutions to the polynomial equation $x^n - 1 = 0$.

7. `QQbar`: the field formed by combining the rationals with the solutions to *every* polynomial equation with integer coefficients. This is known as a the field of algebraic numbers, denoted as $\overline{\mathbb{Q}}$.

8. `FiniteField(p)`: For a prime p, the field of integers $\mathbb{Z}_p$.

If you print a description of some of the above rings, you will sometimes see a new symbol introduced. Consider the following example:

```
sage: F = QuadraticField(7)
sage: F
```

```
Number Field in a with defining polynomial x^2 - 7

sage: root = F.gen(0)
sage: root^2
7

sage: root
a

sage: (2*root)^3
56*a
```

Here `Number Field` describes an object generally formed by combining the rationals with another number (here $\sqrt{7}$). "a" is a new symbol which behaves as a root of the polynomial $x^2 - 7$. We do not say which root, $\sqrt{7}$ or $-\sqrt{7}$, and as we understand the theory better we will see that this does not really matter.

We can obtain this root as a generator of the number field, and then manipulate it. First squaring `root` yields 7. Notice that `root` prints as `a`. Notice, too, that computations with `root` behave as if it was *either* root of $x^2 - 7$, and results print using `a`.

This can get a bit confusing, inputing computations with `root` and getting output in terms of `a`. Fortunately, there is a better way. Consider the following example:

```
sage: F.<b> = QuadraticField(7)
sage: F
Number Field in b with defining polynomial x^2 - 7

sage: b^2
7

sage: (2*b)^3
56*b
```

With the syntax `F.<b>` we can create the field `F` along with specifying a generator `b` using a name of our choosing. Then computations can use `b` in both input and output as a root of $x^2 - 7$.

Here are three new rings that are best created using this new syntax.

1. `F.<a> = FiniteField(p^n)`: We will later have a theorem that tells us that finite fields only exist with orders equal to to a power of a prime. When the power is larger than 1, then we need a gnerator, here given as `a`.

2. `P.<x>=R[]`: the ring of all polynomials in the variable `x`, with coefficients from the ring `R`. Notice that `R` can be *any* ring, so this is a very general construction that uses one ring to form another. See an example below.

3. `Q.<r,s,t> = QuaternionAlgebra(n, m)`: the rationals combined with indeterminates `r`, `s` and `t` such that $r^2 = n$, $s^2 = m$ and $t = rs = -sr$. This is a generalization of the quaternions described in this chapter, though over the rationals rather than the reals, so it is an exact ring. Notice that this is one of the few

noncommutative rings in Sage. The "usual" quaternions would be constructed with `Q.<I,J,K> = QuaternionAlgebra(-1, -1)`. (Notice that using `I` here is not a good choice, because it will then clobber the symbol `I` used for complex numbers.)

Syntax specifying names for generators can be used for many of the above rings as well, such as demonstrated above for quadratic fields and below for cyclotomic fields.

```
sage: C.<t> = CyclotomicField(8)
sage: C.random_element()  # random
-2/11*t^2 + t - 1
```

16.1.2 Properties of Rings

The examples below demonstrate how to query certain properties of rings. If you are playing along, be sure to execute the first compute cell to define the various rings involved in the examples.

```
sage: Z7 = Integers(7)
sage: Z9 = Integers(9)
sage: Q = QuadraticField(-11)
sage: F.<a> = FiniteField(3^2)
sage: P.<x> = Z7[]
sage: S.<f,g,h> = QuaternionAlgebra(-7, 3)
```

Exact versus inexact.

```
sage: QQ.is_exact()
True

sage: RR.is_exact()
False
```

Finite versus infinite.

```
sage: Z7.is_finite()
True

sage: P.is_finite()
False
```

Integral domain?

```
sage: Z7.is_integral_domain()
True

sage: Z9.is_integral_domain()
False
```

Field?

```
sage: Z9.is_field()
False

sage: F.is_field()
True

sage: Q.is_field()
True
```

Commutative?

```
sage: Q.is_commutative()
True

sage: S.is_commutative()
False
```

Characteristic.

```
sage: Z7.characteristic()
7

sage: Z9.characteristic()
9

sage: Q.characteristic()
0

sage: F.characteristic()
3

sage: P.characteristic()
7

sage: S.characteristic()
0
```

Additive and multiplicative identities *print* like you would expect, but notice that while they may *print* identically, they could be *different* because of the ring they are in.

```
sage: b = Z9.zero(); b
0

sage: b.parent()
Ring of integers modulo 9
```

```
sage: c = Q.zero(); c
0

sage: c.parent()
Number Field in a with defining polynomial x^2 + 11

sage: b == c
False

sage: d = Z9.one(); d
1

sage: d.parent()
Ring of integers modulo 9

sage: e = Q.one(); e
1

sage: e.parent()
Number Field in a with defining polynomial x^2 + 11

sage: d == e
False
```

There is some support for subrings. For example, `Q` and `S` are extensions of the rationals, while `F` is totally distinct from the rationals.

```
sage: QQ.is_subring(Q)
True

sage: QQ.is_subring(S)
True

sage: QQ.is_subring(F)
False
```

Not every element of a ring may have a multiplicative inverse, in other words, not every element has to be a unit (unless the ring is a field). It would now be good practice to check if an element is a unit before you try computing its inverse.

```
sage: three = Z9(3)
sage: three.is_unit()
False

sage: three*three
0

sage: four = Z9(4)
sage: four.is_unit()
True

sage: g = four^-1; g
7

sage: four*g
1
```

16.1.3 Quotient Structure

Ideals are the normal subgroups of rings and allow us to build "quotients" — basically new rings defined on equivalence classes of elements of the original ring. Sage support for ideals is variable. When they can be created, there is not always a lot you can do with them. But they work well in certain very important cases. The integers, $\mathbb{Z}$, have ideals that are just multiples of a single integer. We can create them with the `.ideal()` method or just by wrting a scalar multiple of `ZZ`. And then the quotient is isomorphic to a well-understood ring. (Notice that `I` is a bad name for an ideal if we want to work with complex numbers later.)

```
sage: I1 = ZZ.ideal(4)
sage: I2 = 4*ZZ
sage: I3 = (-4)*ZZ
sage: I1 == I2
True

sage: I2 == I3
True

sage: Q = ZZ.quotient(I1); Q
Ring of integers modulo 4

sage: Q == Integers(4)
True
```

We might normally be more careful about the last statement. The quotient is a set of equivalence classes, each infinite, and certainly not a single integer. But the quotient is *isomorphic* to $\mathbb{Z}_4$, so Sage just makes this identification.

```
sage: Z7 = Integers(7)
sage: P.<y> = Z7[]
sage: M = P.ideal(y^2+4)
sage: Q = P.quotient(M)
sage: Q
Univariate Quotient Polynomial Ring in ybar over
Ring of integers modulo 7 with modulus y^2 + 4

sage: Q.random_element() # random
2*ybar + 6

sage: Q.order()
49

sage: Q.is_field()
True
```

Notice that the construction of the quotient ring has created a new generator, converting `y` (y) to `ybar` ($\overline{y}$). We can override this as before with the syntax demonstrated below.

```
sage: Q.<t> = P.quotient(M); Q
Univariate Quotient Polynomial Ring in t over
Ring of integers modulo 7 with modulus y^2 + 4

sage: Q.random_element() # random
4*t + 6
```

So from a quotient of an infinite ring and an ideal (which is also a ring), we create a field, which is finite. Understanding this construction will be an important theme in the next few chapters. To see how remarkable it is, consider what happens with just one little change.

```
sage: Z7 = Integers(7)
sage: P.<y> = Z7[]
sage: M = P.ideal(y^2+3)
sage: Q.<t> = P.quotient(M)
sage: Q
Univariate Quotient Polynomial Ring in t over
Ring of integers modulo 7 with modulus y^2 + 3

sage: Q.random_element() #random
3*t + 1

sage: Q.order()
49

sage: Q.is_field()
False
```

There are a few methods available which will give us properties of ideals. In particular, we can check for prime and maximal ideals in rings of polynomials. Examine the results above and below in the context of Theorem 16.15.

```
sage: Z7 = Integers(7)
sage: P.<y> = Z7[]
sage: M = P.ideal(y^2+4)
sage: N = P.ideal(y^2+3)
sage: M.is_maximal()
True

sage: N.is_maximal()
False
```

The fact that M is a prime ideal is verification of Corollary 16.17.

```
sage: M.is_prime()
True

sage: N.is_prime()
False
```

16.1.4 Ring Homomorphisms

When Sage is presented with 3 + 4/3, how does it know that 3 is meant to be an integer? And then to add it to a rational, how does it know that we really want to view the computation as 3/1 + 4/3? This is really easy for you and me, but devilishly hard for a program, and you can imagine it getting ever more complicated with the many possible rings in Sage, subrings, matrices, etc. Part of the answer is that Sage uses ring homomorphisms to "translate" objects (numbers) between rings.

We will give an example below, but not pursue the topic much further. For the curious, reading the Sage documentation and experimenting would be a good exercise.

```
sage: H = Hom(ZZ, QQ)
sage: phi = H([1])
sage: phi
Ring morphism:
  From: Integer Ring
  To:   Rational Field
  Defn: 1 |--> 1

sage: phi.parent()
Set of Homomorphisms from Integer Ring to Rational Field

sage: a = 3; a
3

sage: a.parent()
Integer Ring

sage: b = phi(3); b
3

sage: b.parent()
Rational Field
```

So phi is a homomorphism ("morphism") that converts integers (the domain is ZZ) into rationals (the codomain is QQ), whose parent is a set of homomorphisms that Sage calls a "homset." Even though a and b both print as 3, which is indistinguishable to our eyes, the parents of a and b are different. Yet the numerical value of the two objects has not changed.

16.2 Exercises

1 Define the two rings $\mathbb{Z}_{11}$ and $\mathbb{Z}_{12}$ with the commands `R = Integers(11)` and `S = Integers(12)`. For each ring, use the relevant command to determine: if the ring is finite, if it is commutative, if it is an integral domain and if it is a field. Then use single Sage commands to find the order of the ring, list the elements, and output the multiplicative identity (i.e. 1, if it exists).

2 Define `R` to be the ring of integers, $\mathbb{Z}$, by executing `R = ZZ` or `R = Integers()`. A command like `R.ideal(4)` will create the principal ideal $\langle 4 \rangle$. The same command can accept more than one generator, so for example, `R.ideal(3, 5)` will create the ideal $\{a \cdot 3 + b \cdot 5 \mid a, b \in \mathbb{Z}\}$. Create several ideals of $\mathbb{Z}$ with two generators and ask Sage to print each as you create it. Explain what you observe.

3 Create a finite field F of order 81 with `F.<t>=FiniteField(3^4)`.
(a) List the elements of F.
(b) Obtain the generators of F with `F.gens()`.
(c) Obtain the first generator of F and save it as `u` with `u = F.0` (alternatively, `u = F.gen(0)`).
(d) Compute the first 80 powers of `u` and comment.
(e) The generator you have worked with above is a root of a polynomial over $\mathbb{Z}_3$. Obtain this polynomial with `F.modulus()` and use this observation to explain the entry in your list of powers that is the fourth power of the generator.

4 Build and analyze a quotient ring as follows:
(a) Use `P.<z>=(Integers(7))[]` to construct a ring P of polynomials in z with coefficients from $\mathbb{Z}_7$.
(b) Use `K = P.ideal(z^+z+3)` to build a principal ideal K generated by the polynomial $z^2 + z + 3$.
(c) Use `H = P.quotient(K)` to build H, the quotient ring of P by K.
(d) Use Sage to verify that H is a field.
(e) As in the previous exercise, obtain a generator and examine the proper collection of powers of that generator.

Chapter 17

Polynomials

17.1 Discussion

Sage is particularly adept at building, analyzing and manipulating polynomial rings. We have seen some of this in the previous chapter. Let's begin by creating three polynomial rings and checking some of their basic properties. There are several ways to construct polynomial rings, but the syntax used here is the most straightforward.

17.1.1 Polynomial Rings and their Elements

```
sage: x, y, z = var('x y z')
sage: R.<x> = Integers(8)[]; R
Univariate Polynomial Ring in x over Ring of integers modulo 8

sage: S.<y> = ZZ[]; S
Univariate Polynomial Ring in y over Integer Ring

sage: T.<z> = QQ[]; T
Univariate Polynomial Ring in z over Rational Field
```

Basic properties of rings are availble for these examples.

```
sage: R.is_finite()
False

sage: R.is_integral_domain()
False

sage: S.is_integral_domain()
True

sage: T.is_field()
False
```

```
sage: R.characteristic()
8

sage: T.characteristic()
0
```

With the construction syntax used above, the variables can be used to create elements of the polynomial ring without explicit coercion (though we need to be careful about constant polynomials).

```
sage: y in S
True

sage: x in S
False

sage: q = (3/2) + (5/4)*z^2
sage: q in T
True

sage: 3 in S
True

sage: r = 3
sage: r.parent()
Integer Ring

sage: s = 3*y^0
sage: s.parent()
Univariate Polynomial Ring in y over Integer Ring
```

Polynomials can be evaluated like they are functions, so we can mimic the evaluation homomorphism.

```
sage: p = 3 + 5*x + 2*x^2
sage: p.parent()
Univariate Polynomial Ring in x over Ring of integers modulo 8

sage: p(1)
2

sage: [p(t) for t in Integers(8)]
[3, 2, 5, 4, 7, 6, 1, 0]
```

Notice that `p` is a degree two polynomial, yet through a brute-force examination we see that the polynomial only has one root, contrary to our usual expectations. It can be even more unusual.

```
sage: q = 4*x^2+4*x
sage: [q(t) for t in Integers(8)]
[0, 0, 0, 0, 0, 0, 0, 0]
```

Sage can create and manipulate rings of polynomials in more than one variable, though we will not have much occasion to use this functionality in this course.

```
sage: s, t = var('s t')
sage: M.<s, t> = QQ[]; M
Multivariate Polynomial Ring in s, t over Rational Field
```

17.1.2 Irreducible Polynomials

Whether or not a polynomial factors, taking into consideration the ring used for its coefficients, is an important topic in this chapter and many of the following chapters. Sage can factor, and determine irreducibility, over the integers, the rationals, and finite fields.

First, over the rationals.

```
sage: x = var('x')
sage: R.<x> = QQ[]
sage: p = 1/4*x^4 - x^3 + x^2 - x - 1/2
sage: p.is_irreducible()
True

sage: p.factor()
(1/4) * (x^4 - 4*x^3 + 4*x^2 - 4*x - 2)

sage: q = 2*x^5 + 5/2*x^4 + 3/4*x^3 - 25/24*x^2 - x - 1/2
sage: q.is_irreducible()
False

sage: q.factor()
(2) * (x^2 + 3/2*x + 3/4) * (x^3 - 1/4*x^2 - 1/3)
```

Factoring over the integers is really no different than factoring over the rationals. This is the content of Theorem 17.9 — finding a factorization over the integers can be converted to finding a factorization over the rationals. So it is with Sage, there is little difference between working over the rationals and the integers. It is a little different working over a finite field. Commentary follows.

```
sage: F.<a> = FiniteField(5^2)
sage: S.<y> = F[]
sage: p = 2*y^5 + 2*y^4 + 4*y^3 + 2*y^2 + 3*y + 1
sage: p.is_irreducible()
True
```

```
sage: p.factor()
(2) * (y^5 + y^4 + 2*y^3 + y^2 + 4*y + 3)

sage: q = 3*y^4+2*y^3-y+4; q.factor()
(3) * (y^2 + (a + 4)*y + 2*a + 3) * (y^2 + 4*a*y + 3*a)

sage: r = y^4+2*y^3+3*y^2+4; r.factor()
(y + 4) * (y^3 + 3*y^2 + y + 1)

sage: s = 3*y^4+2*y^3-y+3; s.factor()
(3) * (y + 1) * (y + 3) * (y + 2*a + 4) * (y + 3*a + 1)
```

To check these factorizations, we need to compute in the finite field, `F`, and so we need to know how the symbol `a` behaves. This symbol is considered as a root of a degree two polynomial over the integers mod 5, which we can get with the `.modulus()` method.

```
sage: F.modulus()
x^2 + 4*x + 2
```

So $a^2+4a+2=0$, or $a^2=-4a-3=a+2$. So when checking the factorizations, anytime you see an a^2 you can replace it by $a+2$. Notice that by Corollary 17.5 we could find the one linear factor of `r`, and the four linear factors of `s`, through a brute-force search for roots. This is feasible because the field is finite.

```
sage: [t for t in F if r(t)==0]
[1]

sage: [t for t in F if s(t)==0]
[2, 2*a + 4, 4, 3*a + 1]
```

However, `q` factors into a pair of degree 2 polynomials, so no amount of testing for roots will discover a factor. With Eisenstein's Criterion, we can create irreducible polynomials, such as in Example 7.

```
sage: W.<w> = QQ[]
sage: p = 16*w^5 - 9*w^4 +3*w^2 + 6*w -21
sage: p.is_irreducible()
True
```

Over the field $\mathbb{Z}_p$, the field of integers mod a prime p, Conway polynomials are canonical choices of a polynomial of degree n that is irreducible over $\mathbb{Z}_p$. See the exercise for more about these polynomials.

17.1.3 Polynomials over Fields

If F is a field, then every ideal of $F[x]$ is principal (Theorem 17.12). Nothing stops you from giving Sage two (or more) generators to construct an ideal, but Sage will determine the element to use in a description of the ideal as a principal ideal.

```
sage: W.<w> = QQ[]
sage: r = -w^5 + 5*w^4 - 4*w^3 + 14*w^2 - 67*w + 17
sage: s = 3*w^5 - 14*w^4 + 12*w^3 - 6*w^2 + w
sage: S = W.ideal(r, s)
sage: S
Principal ideal (w^2 - 4*w + 1) of
Univariate Polynomial Ring in w over Rational Field

sage: (w^2)*r + (3*w-6)*s in S
True
```

Theorem 17.13 is the key fact that allows us to easily construct finite fields. Here is a construction of a finite field of order $7^5 = 16\,807$. All we need is a polynomial of degree 5 that is irreducible over $\mathbb{Z}_7$.

```
sage: F = Integers(7)
sage: R.<x> = F[]
sage: p = x^5+ x + 4
sage: p.is_irreducible()
True

sage: id = R.ideal(p)
sage: Q = R.quotient(id); Q
Univariate Quotient Polynomial Ring in xbar over
Ring of integers modulo 7 with modulus x^5 + x + 4

sage: Q.is_field()
True

sage: Q.order() == 7^5
True
```

The symbol `xbar` is a generator of the field, but right now it is not accessible. `xbar` is the coset $x + \langle x^5 + x + 4\rangle$. A better construction would include specifying this generator.

```
sage: Q.gen(0)
xbar

sage: Q.<t> = R.quotient(id); Q
Univariate Quotient Polynomial Ring in t over
Ring of integers modulo 7 with modulus x^5 + x + 4
```

```
sage: t^5 + t + 4
0

sage: t^5 == -(t+4)
True

sage: t^5
6*t + 3

sage: (3*t^3 + t + 5)*(t^2 + 4*t + 2)
5*t^4 + 2*t^2 + 5*t + 5

sage: a = 3*t^4 - 6*t^3 + 3*t^2 + 5*t + 2
sage: ainv = a^-1; ainv
6*t^4 + 5*t^2 + 4

sage: a*ainv
1
```

17.2 Exercises

1 Consider the polynomial $x^3 - 3x + 4$. Compute the most thorough factorization of this polynomial over each of the following fields: (a) the finite field $\mathbb{Z}_5$, (b) a finite field with 125 elements, (c) the rationals, (d) the real numbers and (e) the complex numbers. To do this, build the appropriate polynomial ring, and construct the polynomial as a member of this ring, and use the `.factor()` method.

2 "Conway polynomials" are irreducible polynomials over $\mathbb{Z}_p$ that Sage (and other software) uses to build maximal ideals in polynomial rings, and thus quotient rings that are fields. Roughly speaking, they are "canonical" choices for each degree and each prime. The command `conway_polynomial(p, n)` will return a database entry that is an irreducible polynomial of degree n over $\mathbb{Z}_p$.

Execute the command `conway_polynomial(5, 4)` to obtain an allegedly irreducible polynomial of degree 4 over $\mathbb{Z}_5$: $p = x^4 + 4x^2 + 4x + 2$. First determine that p has no linear factors. The only possibility left is that `p` factors as two quadratic polynomials over $\mathbb{Z}_5$. Use a list comprehension with *three* `for` statements to create *every* possible quadratic polynomial over $\mathbb{Z}_5$. Now use this list to create every possible product of two quadratic polynomials and check to see if `p` is in this list.

More on Conway polynomials is available at http://www.math.rwth-aachen.de/~Frank.Luebeck/data/ConwayPol/index.html

3 Construct a finite field of order 729 as a quotient of a polynomial ring by a principal ideal generated with a Conway polynomial.

4 Define the polynomials $p = x^3 + 2x^2 + 2x + 4$ and $q = x^4 + 2x^2$ as polynomials with coefficients from the integers. Compute `gcd(p, q)` and verify that the result divides both `p` and `q` (just form a fraction in Sage and see that it simplifies cleanly). Now compute the extended gcd, `xgcd(p, q)`. Verify divisibility and the property that makes this an extended gcd. Can you explain why there are two different results for the gcd? Why does this not violate Proposition 17.7?

5 For a polynomial ring over a field, every ideal is principal. Begin with the ring of polynomials over the rationals. Experiment with constructing ideals using two generators and then see that Sage converts the ideal to a principal ideal with a single generator. (You can get this generator with the ideal method `.gen()`.) Can you explain how this single generator is computed?

Chapter 18

Integral Domains

18.1 Discussion

We have already seen some integral domains and unique factorizations in the previous two chapters. In addition to what we have already seen, Sage has support for some of the topics from this section, but the coverage is limited. Some functions will work for some rings and not others, while some functions are not yet part of Sage. So we will give some examples, but this is far from comprehensive.

18.1.1 Field of Fractions

Sage is frequently able to construct a field of fractions, or identify a certain field as the field of fractions. For example, the ring of integers and the field of rational numbers are both implemented in Sage, and the integers "know" that the rationals is it's field of fractions.

```
sage: Q = ZZ.fraction_field(); Q
Rational Field

sage: Q == QQ
True
```

In other cases Sage will construct a fraction field, in the spirit of Lemma 18.3. So it is then possible to do basic calculations in the constructed field.

```
sage: R.<x> = ZZ[]
sage: P = R.fraction_field();P
Fraction Field of Univariate Polynomial Ring in x over Integer Ring

sage: f = P((x^2+3)/(7*x+4))
sage: g = P((4*x^2)/(3*x^2-5*x+4))
sage: h = P((-2*x^3+4*x^2+3)/(x^2+1))
sage: ((f+g)/h).numerator()
3*x^6 + 23*x^5 + 32*x^4 + 8*x^3 + 41*x^2 - 15*x + 12

sage: ((f+g)/h).denominator()
-42*x^6 + 130*x^5 - 108*x^4 + 63*x^3 - 5*x^2 + 24*x + 48
```

18.1.2 Prime Subfields

Corollary 18.6 says every field of characteristic p has a subfield isomorphic to $\mathbb{Z}_p$. For a finite field, the exact nature of this subfield is not a surprise, but Sage will allow us to extract it easily.

```
sage: F.<c> = FiniteField(3^5)
sage: F.characteristic()
3

sage: G = F.prime_subfield(); G
Finite Field of size 3

sage: G.list()
[0, 1, 2]
```

More generally, the fields mentioned in the conclusions of Corollary 18.5 and Corollary 18.6 are known as the "prime subfield" of the ring containing them. Here is an example of the characteristic zero case.

```
sage: K.<y>=QuadraticField(-7); K
Number Field in y with defining polynomial x^2 + 7

sage: K.prime_subfield()
Rational Field
```

In a rough sense, every characteristic zero field contains a copy of the rational numbers (the fraction field of the integers), which can explain Sage's extensive support for rings and fields that extend the integers and the rationals.

18.1.3 Integral Domains

Sage can determine if some rings are integral domains and we can test products in them. However, notions of units, irreducibles or prime elements are not generally supported (outside of what we have seen for polynomials in the previous chapter). Worse, the construction below creates a ring within a larger field and so some functions (such as `.is_unit()`) pass through and give misleading results. This is because the construction below creates a ring known as an "order in a number field."

Note also in the following example that $\mathbb{Z}[\sqrt{3}i]$ is built with *two* generators, and there is no easy way to get the internal and external names of the generators to synchronize.

```
sage: K.<x,y> = ZZ[sqrt(-3)]; K
Order in Number Field in a with defining polynomial x^2 + 3

sage: K.is_integral_domain()
True
```

```
sage: K.gens()
[1, a]

sage: (x, y)
(1, a)

sage: (1+y)*(1-y) == 2*2
True
```

The following is misleading, since 4, as an element of $\mathbb{Z}[\sqrt{3}i]$ does not have a multiplicative inverse. Here the computations are being performed in a bigger ring.

```
sage: four = K(4)
sage: four.is_unit()
True

sage: four^-1
1/4
```

18.1.4 Principal Ideals

When a ring is a principal ideal domain, such as the integers, or polynomials over a field, Sage works well. Beyond that, support begins to weaken.

```
sage: T.<x>=ZZ[]
sage: T.is_integral_domain()
True

sage: J = T.ideal(5, x); J
Ideal (5, x) of Univariate Polynomial Ring in x over Integer Ring

sage: Q = T.quotient(J); Q
Quotient of Univariate Polynomial Ring in x over
Integer Ring by the ideal (5, x)

sage: J.is_principal()
Traceback (most recent call last):
...
NotImplementedError

sage: Q.is_field()
Traceback (most recent call last):
...
NotImplementedError
```

18.2 Exercises

There are no exercises for this section.

Chapter 19

Lattices and Boolean Algebras

19.1 Discussion

Sage has support for both partially ordered sets ("posets") and lattices, and does an excellent job of providing visual depictions of both.

19.1.1 Creating Partially Ordered Sets

Example 5 in the text is a good example to replicate as a demonstration of Sage commands. We first define the elements of the set X.

```
sage: X = [1, 2, 3, 4, 6, 8, 12, 24]
```

One approach to creating the relation is to specify *every* instance where one element is comparable to the another. So we build a list of pairs, where each pair contains comparable elements, with the lesser one first. This is the set of relations.

```
sage: R = [(a,b) for a in X for b in X if a.divides(b)]; R
[(1, 1), (1, 2), (1, 3), (1, 4), (1, 6), (1, 8), (1, 12), (1, 24),
 (2, 2), (2, 4), (2, 6), (2, 8), (2, 12), (2, 24), (3, 3), (3, 6),
 (3, 12), (3, 24), (4, 4), (4, 8), (4, 12), (4, 24), (6, 6),
 (6, 12), (6, 24), (8, 8), (8, 24), (12, 12), (12, 24), (24, 24)]
```

We construct the poset by giving the the `Poset` constructor a list containing the elements and the relations. We can then easily get a "plot" of the poset. Notice the plot just shows the "cover relations" — a minimal set of comparisons which the assumption of transitivity would expand into all the relations.

```
sage: D = Poset([X, R])
sage: D.plot()    # not tested
```

Another approach to creating a `Poset` is to let the poset constructor run over all the pairs of elements, and all we do is give the constructor a way to test if two elements are comparable. Our comparison function should expect two elements and then return `True` or `False`. A "lambda" function is one way to quickly build such a

function. This may be a new idea for you, but mastering lambda functions can be a great convenience. Notice that "lambda" is a word reserved for just this purpose. There are other ways to make functions in Sage, but a lambda function is quickest when the function is simple.

```
sage: divisible = lambda x, y: x.divides(y)
sage: L = Poset([X, divisible])
sage: L == D
True

sage: L.plot()    # not tested
```

Sage also has a collection of stock posets. Some are one-shot constructions, while others are members of parameterized families. Use tab-completion on `Posets.` to see the full list. Here are some examples.

A one-shot construction. Perhaps what you would expect, though there might be other, equally plausible, alternatives.

```
sage: Q = Posets.PentagonPoset()
sage: Q.plot()    # not tested
```

A parameterized family. This is the classic example where the elements are subsets of a set with n elements and the relation is "subset of."

```
sage: S = Posets.BooleanLattice(4)
sage: S.plot()    # not tested
```

And random posets. These can be useful for testing and experimenting, but are unlikely to exhibit special cases that may be important. You might run the following command many times and vary the second argument, which is a rough upper bound on the probability any two elements are comparable. Remember that the plot only shows the cover relations. The more elements that are comparable, the more "vertically stretched" the plot will be.

```
sage: T = Posets.RandomPoset(20,0.05)
sage: T.plot()    # not tested
```

19.1.2 Properties of a Poset

Once you have a poset, what can you do with it? Let's return to our first example, `D`. We can of course determine if one element is less than another, which is the fundamental structure of a poset.

```
sage: D.is_lequal(4, 8)
True

sage: D.is_lequal(4, 4)
True
```

```
sage: D.is_less_than(4, 8)
True

sage: D.is_less_than(4, 4)
False

sage: D.is_lequal(6, 8)
False

sage: D.is_lequal(8, 6)
False
```

Notice that `6` and `8` are not comparable in this poset (it is a *partial* order). The methods `.is_gequal()` and `.is_greater_than()` work similarly, but returns `True` if the first element is greater (or equal).

```
sage: D.is_gequal(8, 4)
True

sage: D.is_greater_than(4, 8)
False
```

We can find the largest and smallest elements of a poset. This is a random poset built with a 10% probability, but copied here to be repeatable.

```
sage: X = range(20)
sage: C = [[18, 7],  [9, 11], [9, 10], [11, 8], [6, 10],
...        [10, 2],   [0, 2],  [2, 1],  [1, 8], [8, 12],
...         [8, 3],  [3, 15], [15, 7], [7, 16],  [7, 4],
...       [16, 17], [16, 13], [4, 19], [4, 14], [14, 5]]
sage: P = Poset([X, C])
sage: P.plot()    # not tested

sage: P.minimal_elements()
[18, 9, 6, 0]

sage: P.maximal_elements()
[17, 13, 19, 5, 12]
```

Elements of a poset can be partioned into level sets. In plots of posets, elements at the same level are plotted vertically at the same height. Each level set is obtained by removing all of the previous level sets and then taking the minimal elements of the result.

```
sage: P.level_sets()
[[18, 9, 6, 0], [11, 10], [2], [1], [8], [3, 12],
 [15], [7], [16, 4], [17, 13, 19, 14], [5]]
```

If we make two elements in R comparable when they had not previously been, this is an extension of R. Consider all possible extensions of one poset — we can make a poset from all of these, where set inclusion is the relation. A linear extension is a maximal element in this poset of posets. Informally, we are adding as many new relations as possible, consistent with the original poset and so that the result is a total order (there is an ordering of the elements consistent with the order in the poset). We can build such a thing, but the output is just a list of the elements in the linear order. A computer scientist would be inclined to call this a "topological sort."

```
sage: linear = P.linear_extension(); linear
[18, 9, 11, 6, 10, 0, 2, 1, 8, 3, 15,
  7, 16, 17, 13, 4, 19, 14, 5, 12]
```

We can construct subposets by giving a set of elements to induce the new poset. Here we take roughly the "bottom half" of the random poset P by inducing the subposet on a union of some of the level sets.

```
sage: level = P.level_sets()
sage: bottomhalf = sum([level[i] for i in range(5)], [])
sage: B = P.subposet(bottomhalf)
sage: B.plot()    # not tested
```

The dual of a poset retains the same set of elements, but reverses any comparisons.

```
sage: Pdual = P.dual()
sage: Pdual.plot()    # not tested
```

Taking the dual of the divisibility poset from Example 5 would be like changing the relation to "is a multiple of."

```
sage: Ddual = D.dual()
sage: Ddual.plot()    # not tested
```

19.1.3 Lattices

Every lattice is a poset, so all the commands above will work equally well for a lattice. But how do you create a lattice? Simple — first create a poset and then feed it into the LatticePoset() constructor. But realize that just because you give this constructor a poset, it does not mean a lattice will always come back out. Only if the poset *is already* a lattice will it get upgraded from a poset to a lattice for Sage's purposes.

An integer composition of n is an ordered list of positive integers that sum to n. One composition covers another if it can be formed by adding two consecutive parts of the larger composition, and possibly re-sorting. For example, $[2, 1, 2] > [3, 2]$. This forms a poset that is also a lattice.

```
sage: CP = Posets.IntegerCompositions(5)
sage: C = LatticePoset(CP)
sage: C.plot()    # not tested
```

A meet or a join is a fundamental operation in a lattice.

```
sage: C.meet([1,1,1,2], [2,1,1,1])
[2, 1, 2]

sage: C.join([1,4], [2,3])
[1, 1, 3]
```

Once a poset is upgraded to lattice status, then additional commands become available, or the character of their results changes.

An example of the former is the `.is_distributive()` method.

```
sage: C.is_distributive()
True
```

An example of the latter is the `.top()` method. What your text calls a largest element and a smallest element of a lattice, Sage calls a top and a bottom. For a poset, `.top()` and `.bottom()` may return an element or may not (returning `None`), but for a lattice it is guaranteed to return an element.

```
sage: C.top()
[1, 1, 1, 1, 1]

sage: C.bottom()
[5]
```

Notice that the returned values are elements of the lattice, in this case ordered lists of integers summing to 5.

Complements now make sense in a lattice, and the lattice of integer compositions is a complemented lattice.

```
sage: comp = C.complements()
sage: C[comp[2]]
[1, 1, 1, 2]

sage: C[2]
[4, 1]

sage: C.is_complemented()
True
```

There are many more commands which apply to posets and lattices, so build a few and use tab-completion liberally to explore. There is more to discover than we can cover in just a single chapter, but you now have the basic tools to profitably study posets and lattices in Sage.

19.2 Exercises

1 Use `R = Posets.RandomPoset(30,0.05)` to construct a random poset. Use `R.plot()` to get an idea of what you have built.
(a) Illustrate the use of the poset methods
`.is_lequal()`
`.is_less_than()`
`.is_gequal()`
`.is_greater_than()`
to determine if two elements are related or incomparable.
(b) Use `.minimal_elements()` and `,maximal_elements()` to find the smallest and largest elements of your poset.
(c) Use `LatticePoset(R)` to see if the poset `R` is a lattice by attempting to convert it into a lattice.
(d) Find a linear extension of your poset. Confirm that consecutive elements of the output are comparable in the orginal lattice, and that they compare properly.

2 Construct the poset on the positive divisors of $72 = 2^3 \cdot 3^2$ with divisiblity as the relation, and then convert to a lattice.
(a) Determine the one and zero element using `.top()` and `.bottom()`.
(b) Determine all the pairs of elements of the lattice that are complements of each other *without* using the `.complement()` method, but rather just use the `.meet()` and `.join()` methods. Extra credit if you can output each pair just once.
(c) Determine if the lattice is distributive using just the `.meet()` and `.join()` methods, and not the `.is_distributive()` method.

3 Construct several diamond lattices with `Posets.DiamondPoset(n)` by varying the value of `n`. Give answers, with justifications, to these questions for *general* n, based on observations obtained from experiments with Sage.
(a) Which elements have complements and which do not, and why?
(b) Read the documentation of the `.antichains()` method to learn what an antichain is. How many antichains are there?
(c) Is the lattice distributive?

4 Use `Posets.BooleanLattice(4)` to construct an instance of the prototypical Boolean algebra on 16 elements (i.e. all subsets of a 4-set).

Then use `Posets.IntegerCompositions(5)` to construct the poset whose 16 elements are the compositions of the integer 5. We have seen above that the integer composition lattice is distributive and complemented, making it a Boolean algebra.

And by Theorem 19.12 we can conclude that these two Boolean algebras are isomorphic.

Plot each to see the similarity, as follows. Use the method `.hasse_diagram()` on each poset to get back a directed graph and ask for their plots. Then use the graph method `.is_isomorphic()` to see that the two Hasse diagrams really are the same.

5 (Advanced) For the previous question, construct an explicit isomorphism between the two Boolean algebras. This would be a bijective function (constructed with the `def` command) that converts compositions into sets (or if, you choose, sets into compositions) and which respects the meet and join operations. You can test and illustrate your function by its interaction with specific elements evaluated in the meet and join operations, as described in the definition of an isomorphism of Boolean algebras.

Chapter 20

Vector Spaces

20.1 Discussion

Many computations, in seemingly very different areas of mathematics, can be translated into questions about linear combinations, or other areas of linear algebra. So Sage has extensive and thorough support for topics such as vector spaces.

20.1.1 Vector Spaces

The simplest way to create a vector space is to begin with a field and write a "power" to indicate the number of entries in the vectors of the space.

```
sage: V = QQ^4; V
Vector space of dimension 4 over Rational Field

sage: F.<a> = FiniteField(3^4)
sage: W = F^5; W
Vector space of dimension 5 over Finite Field in a of size 3^4
```

Elements can be built with the vector constructor.

```
sage: v = vector(QQ, [1, 1/2, 1/3, 1/4]); v
(1, 1/2, 1/3, 1/4)

sage: v in V
True

sage: w = vector(F, [1, a^2, a^4, a^6, a^8]); w
(1, a^2, a^3 + 1, a^3 + a^2 + a + 1, a^2 + a + 2)

sage: w in W
True
```

Notice that vectors are printed with parentheses, which helps distinguish them from lists. Vectors print horizontally, but in Sage there is no such thing as a "row vector" or a "column vector," though once matrices get involved we need to address this distinction. Finally, notice how the elements of the finite field have been converted to an alternate representation. Once we have vector spaces full of vectors, we can perform computations with them. Ultimately, all the action in a vector space comes back to vector addition and scalar multiplication, which together create linear combinations.

```
sage: u = vector(QQ, [ 1, 2,  3, 4,   5,  6])
sage: v = vector(QQ, [-1, 2, -4, 8, -16, 32])
sage: 3*u - 2*v
(5, 2, 17, -4, 47, -46)

sage: w = vector(F, [1, a^2, a^4, a^6,  a^8])
sage: x = vector(F, [1,   a, 2*a,   a,    1])
sage: y = vector(F, [1, a^3, a^6, a^9, a^12])
sage: a^25*w + a^43*x + a^66*y
(a^3 + a^2 + a + 2, a^2 + 2*a, 2*a^3 + a^2 + 2, 2*a^3 + a^2 + a,
 a^3 + 2*a^2 + a + 2)
```

20.1.2 Subspaces

Sage can create subspaces in a variety of ways, such as in the creation of row or column spaces of matrices. However, the most direct way is to begin with a set of vectors to use as a spanning set.

```
sage: u = vector(QQ, [1, -1, 3])
sage: v = vector(QQ, [2, 1, -1])
sage: w = vector(QQ, [3, 0, 2])
sage: S = (QQ^3).subspace([u, v, w]); S
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0  2/3]
[   0    1 -7/3]

sage: 3*u - 6*v + (1/2)*w in S
True

sage: vector(QQ, [4, -1, -2]) in S
False
```

Notice that the information printed about `S` includes a "basis matrix." The rows of this matrix are a basis for the vector space. We can get the basis, as a list of vectors (not rows of a matrix), with the `.basis()` method.

```
sage: S.basis()
[
(1, 0, 2/3),
(0, 1, -7/3)
]
```

Notice that Sage has converted the spanning set of three vectors into a basis with two vectors. This is partially due to the fact that the original set of three vectors is linearly dependent, but a more substantial change has occurred.

This is a good place to discuss some of the mathematics behind what makes Sage work. A vector space over an infinite field, like the rationals or the reals, is an infinite set. No matter how expansive computer memory may seem, it is still finite. How does Sage fit an infinite set into our finite machines? The main idea is that a finite-dimensional vector space has a finite set of generators, which we know as a basis. So Sage really only needs the elements of a basis (two vectors in the previous example) to be able to work with the infinitely many possibilities for elements of the subspace.

Furthermore, for every basis associated with a vector space, Sage performs linear combinations to convert the given into another "standard" basis. This new basis has the property that as the rows of a matrix, the matrix is in reduced row-echelon form. You can see this in the basis matrix above. The reduced row-echelon form of a matrix is unique, so this standard basis allows Sage to recognize when two vector spaces are equal. Here is an example.

```
sage: u = vector(QQ, [1, -1,  3])
sage: v = vector(QQ, [2,  1, -1])
sage: w = vector(QQ, [3,  0,  2])
sage: u + v == w
True

sage: S1 = (QQ^3).subspace([u, v, w])
sage: S2 = (QQ^3).subspace([u-v, v-w, w-u])
sage: S1 == S2
True
```

As you might expect, it is easy to determine the dimension of a vector space.

```
sage: u = vector(QQ, [1, -1,  3,  4])
sage: v = vector(QQ, [2,  1, -1, -2])
sage: S = (QQ^4).subspace([u, v, 2*u+3*v, -u+2*v])
sage: S.dimension()
2
```

20.1.3 Linear Independence

There are a variety of ways in Sage to determine if a set of vectors is linearly independent or not, and to find relations of linear dependence if they exist. The technique we

will show here is a simple test to see if a set of vectors is linearly independent or not. Simply use the vectors as a spanning set for a subspace, and check the dimension of the subspace. The dimension equals the number of vectors in the spanning set if and only if the spanning set is linearly independent.

```
sage: F.<a> = FiniteField(3^4)
sage: u = vector(F, [a^i for i in range(0,  7, 1)])
sage: v = vector(F, [a^i for i in range(0, 14, 2)])
sage: w = vector(F, [a^i for i in range(0, 21, 3)])
sage: S = (F^7).subspace([u, v, w])
sage: S.dimension()
3

sage: S = (F^7).subspace([u, v, a^3*u + a^11*v])
sage: S.dimension()
2
```

So the first set of vectors, `[u, v, w]`, is linearly independent, while the second set, `[u, v, a^3*u + a^11*v]`, is not.

20.1.4 Abstract Vector Spaces

Sage does not implement many abstract vector spaces directly, such as P_n, the vector space of polynomials of degree n or less. This is due in part to the fact that a finite-dimensional vector space over a field F is isomorphic to the vector space F^n. So Sage captures all the functionality of finite-dimensional vector spaces, and it is left to the user to perform the conversions according to the isomorphism (which is often trivial with the choice of an obvious basis).

However, there are instances where rings behave naturally as vector spaces and we can exploit this extra structure. We will see much more of this in the chapters on fields and Galois theory. As an example, finite fields have a single generator, and the first few powers of the generator form a basis. Consider creating a vector space from the elements of a finite field of order $7^6 = 117\,649$. As elements of a field we know they can be added, so we will *define* this to be the addition in our vector space. For any element of the integers mod 7, we can multiply an element of the field by the integer, so we *define* this to be our scalar multiplication. Later, we will be certain that these two definitions lead to a vector space, but take that for granted now. So here are some operations in our new vector space.

```
sage: F.<a> = FiniteField(7^6)
sage: u = 2*a^5 + 6*a^4 + 2*a^3 + 3*a^2 + 2*a + 3
sage: v = 4*a^5 + 4*a^4 + 4*a^3 + 6*a^2 + 5*a + 6
sage: u + v
6*a^5 + 3*a^4 + 6*a^3 + 2*a^2 + 2

sage: 4*u
a^5 + 3*a^4 + a^3 + 5*a^2 + a + 5
```

```
sage: 2*u + 5*v
3*a^5 + 4*a^4 + 3*a^3 + a^2 + a + 1
```

You might recognize that this looks very familiar to how we add polynomials, and multiply polynomials by scalars. You would be correct. However, notice that in this vector space construction, we are totally ignoring the possibility of multiplying two field elements together. As a vector space with scalars from $\mathbb{Z}_7$, a basis is the first six powers of the generator, $\{1, a, a^2, a^3, a^4, a^5\}$. (Notice how counting from zero is natural here.) You may have noticed how Sage consistently rewrites elements of fields as linear combinations — now you know why.

Here is what Sage knows about a finite field as a vector space. First, it knows that the finite field *is* a vector space, and what the field of scalars is.

```
sage: V = F.vector_space(); V
Vector space of dimension 6 over Finite Field of size 7

sage: R = V.base_ring(); R
Finite Field of size 7

sage: R == FiniteField(7)
True

sage: V.dimension()
6
```

So the finite field (as a vector space) is isomorphic to the vector space $(\mathbb{Z}_7)^6$. Notice this is not a ring or field isomorphism, as it does not fully address multiplication of elements, even though that is possible in the field.

Second, elements of the field can be converted to elements of the vector space easily.

```
sage: u = 2*a^5 + 6*a^4 + 2*a^3 + 3*a^2 + 2*a + 3
sage: v = 4*a^5 + 4*a^4 + 4*a^3 + 6*a^2 + 5*a + 6
sage: x = V(u); x
(3, 2, 3, 2, 6, 2)

sage: y = V(v); y
(6, 5, 6, 4, 4, 4)
```

Notice that Sage writes field elements with high powers of the generator first, while the basis in use is ordered with low powers first. The computations below illustrate the isomorphism between the finite field and $(\mathbb{Z}_7)^6$.

```
sage: F.<a> = FiniteField(7^6)
sage: V = F.vector_space()
sage: R = V.base_ring()
sage: u = 2*a^5 + 6*a^4 + 2*a^3 + 3*a^2 + 2*a + 3
sage: v = 4*a^5 + 4*a^4 + 4*a^3 + 6*a^2 + 5*a + 6
sage: V(u + v) == V(u) + V(v)
True
```

```
sage: two = R(2)
sage: V(two*u) == two*V(u)
True
```

20.1.5 Linear Algebra

Sage has extensive support for linear algebra, well beyond what we have described here, or what we will need for the remaining chapters. Create vector spaces and vectors (with different fields of scalars), and then use tab-completion on these objects to explore the large sets of available commands.

20.2 Exercises

1 Given two subspaces U and W of a vector space V, their sum $U+W$ can be defined as $U+W = \{u+w \mid u \in U,\ w \in W\}$, in other words, the set of all possible sums of an element from U and an element from W.

Notice this is not the direct sum of your text, nor the `direct_sum()` method in Sage. However, you can build this subspace in Sage as follows. Grab the bases of U and W individually, as lists of vectors. Join the two lists together by just using a plus sign between them. Now build the sum subspace by creating a subspace of V spanned by this set, by using the `.subspace()` method.

Build a largish vector space over the rationals (`QQ`), where "largish" means perhaps dimension 7 or 8 or so. Construct a few subspaces and compare their individual dimensions with the dimensions of the intersection of U and W ($U \cap W$, `.intersection()` in Sage) and the sum $U+V$. Form a conjecture relating these dimensions based on your (nontrivial) experiments.

2 We can construct a field that extends the rationals by adding in a fourth-root of two, $\mathbb{Q}[\sqrt[4]{2}]$, in Sage with the command `F.<c> = QQ[2^(1/4)]`. This is a vector space of dimension 4 over the rationals, with a basis that is the first four powers of $c = \sqrt[4]{2}$ (starting with the zero power).

The command `F.vector_space()` will return three items. The first is a vector space over the rationals that is isomorphic to `F`. The next two are isomorphisms between the two vector spaces (one in each direction). These two isomorphisms can then be used like functions. Notice that this is different behavior than for the same command applied to finite fields. Create non-trivial examples that show that these vector space isomorphisms behave as an isomorphism should. (You will have at least four such examples in a complete solution.)

3 Build a finite field F of order p^n in the usual way. Then construct the (multiplicative) group of all invertible (nonsingular) $m \times m$ matrices over this field with the command `G = GL(m, F)` ("the general linear group"). What is the order of this group? In other words, what is a general expression for the order of this group? So your answer should be a function of m, p and n and should include an explanation of how you come by your formula (i.e. something resembling a proof).

Hints: `G.order()` will help you test and verify your hypotheses. Small examples in Sage (listing all the elements of the group) might aid your intuition—which is why this is a Sage exercise. Small means 2×2 and 3×3 matrices and finite fields with $2, 3, 4, 5$ elements, at most. Results don't really depend on each of p and n, but rather just on p^n.

Realize this group is interesting because it contains representations of all the invertible (i.e. 1-1 and onto) linear transformations from the (finite) vector space F^m to itself.

4 What happens if we try to do linear algebra over a *ring* that is not also a *field*? The object that resembles a vector space, but with this one distinction, is known as a "module." You can build one easily with a construction like `ZZ^3`. Evaluate the following to create a module and a submodule.

```
sage: M = ZZ^3
sage: u = M([1, 0, 0])
sage: v = M([2, 2, 0])
sage: w = M([0, 0, 4])
sage: N = M.submodule([u, v, w])
```

Examine the bases and dimensions (aka "rank") of the module and submodule, and check the equality of the module and submodule. How is this different than the situation for vector spaces? Can you create a third module, `P`, that is a proper subset of `M` and properly contains `N`?

5 A finite field, F, of order 5^3 is a vector space of dimension 3 over $\mathbb{Z}_5$. Suppose a is a generator of F. Let M be any 3×3 matrix with entries from $\mathbb{Z}_5$. If we convert an element $x \in F$ to a vector (relative to the basis $\{1, a, a^2\}$), then we can multiply it by M (with M on the left) to create another vector, which we can interpret as a linear combination of the basis elements, and hence another element of F. This function is a vector space homomorphism, better known as a linear transformation. Read each of the three parts below and give an example in each part that does not qualify as an example in the subsequent parts.

(a) Create a "random" matrix M and give examples to show that the mapping described is a vector space homomorphism of F into F.

(b) Create an invertible matrix M. The mapping will now be an invertible homomorphism. Determine the inverse function and give examples to verify its properties.

(c) Since a is a generator of the field, the mapping $a \mapsto a^5$ can be extended to a vector space homomorphism (i.e. a linear transformation). Find a matrix M which effects this linear transformation, and from this, determine that the homomorphism is invertible.
(d) None of the previous three parts applies to properties of multiplication in the field. However, the mapping from part (c) also preserves multiplication in the field, though a proof of this may not be obvious right now. So we are saying this mapping is a field automorphism, preserving both addition and multiplication. Give a nontrivial example of the multiplication-preserving properties of this mapping.

Chapter 21

Fields

21.1 Discussion

In Sage, and other places, an extension of the rationals is called a "number field." They are one of Sage's most mature features.

21.1.1 Number Fields

There are several ways to create a number field. We are familiar with the syntax where we adjoin an irrational number that we can write with traditional combinations of arithmetic and roots.

```
sage: M.<a> = QQ[sqrt(2)+sqrt(3)]; M
Number Field in a with defining polynomial x^4 - 10*x^2 + 1
```

We can also specify the element we want to adjoin as the root of a monic irreducible polynomial. One approach is to construct the polynomial ring first so that the polynomial has the location of its coefficients specified properly.

```
sage: F.<y> = QQ[]
sage: p = y^3 - 1/4*y^2 - 1/16*y + 1/4
sage: p.is_irreducible()
True

sage: N.<b> = NumberField(p, 'b'); N
Number Field in b with
defining polynomial y^3 - 1/4*y^2 - 1/16*y + 1/4
```

Rather than building the whole polynomial ring, we can simply introduce a variable as the generator of a polynomial ring and then create polynomials from this variable. This spares us naming the polynomial ring. Notice in the example that both instances of `z` are necessary.

```
sage: z = polygen(QQ, 'z')
sage: q = z^3 - 1/4*z^2 - 1/16*z + 1/4
sage: q.parent()
Univariate Polynomial Ring in z over Rational Field

sage: P.<c> = NumberField(q, 'c'); P
Number Field in c with
defining polynomial z^3 - 1/4*z^2 - 1/16*z + 1/4
```

We can recover the polynomial used to create a number field, even if we constructed it by giving an expression for an irrational element. In this case, the polynomial is the minimal polynomial of the element.

```
sage: M.polynomial()
x^4 - 10*x^2 + 1

sage: N.polynomial()
y^3 - 1/4*y^2 - 1/16*y + 1/4
```

For any element of a number field, Sage will obligingly compute its minimal polynomial.

```
sage: element = -b^2 + 1/3*b + 4
sage: element.parent()
Number Field in b with
defining polynomial y^3 - 1/4*y^2 - 1/16*y + 1/4

sage: r = element.minpoly('t'); r
t^3 - 571/48*t^2 + 108389/2304*t - 13345/216

sage: r.parent()
Univariate Polynomial Ring in t over Rational Field

sage: r.subs(t=element)
0
```

Substituting `element` back into the alleged minimal polynomial and getting back zero is not convincing eveidence that it is the *minimal* polynomial, but it is heartening.

21.1.2 Relative and Absolute Number Fields

With Sage we can adjoin several elements at once and we can build nested towers of number fields. Sage uses the term "absolute" to refer to a number field viewed as an extension of the rationals themselves, and the term "relative" to refer to a number field constructed, or viewed, as an extension of another (nontrivial) number field.

```
sage: A.<a,b> = QQ[sqrt(2), sqrt(3)]
sage: A
Number Field in sqrt2 with defining polynomial x^2 - 2 over
its base field

sage: B = A.base_field(); B
Number Field in sqrt3 with defining polynomial x^2 - 3

sage: A.is_relative()
True

sage: B.is_relative()
False
```

The number field `A` has been constructed mathematically as what we would write as $\mathbb{Q} \subset \mathbb{Q}[\sqrt{3}] \subset \mathbb{Q}[\sqrt{3}, \sqrt{2}]$. Notice the slight difference in ordering of the elements we are adjoining, and notice how the number fields use slightly fancier internal names (`sqrt2`, `sqrt3`) for the new elements.

We can "flatten" a relative field to view it as an absolute field, which may have been our intention from the start. Here we create a new number field from `A` that makes it a pure absolute number field.

```
sage: C.<c> = A.absolute_field()
sage: C
Number Field in c with defining polynomial x^4 - 10*x^2 + 1
```

Once we construct an absolute number field this way, we can recover isomorphisms to and from the absolute field. Recall that our tower was built with generators `a` and `b`, while the flattened tower is generated by `c`. The `.structure()` method returns a pair of functions, with the absolute number field as the domain and codomain (in that order).

```
sage: fromC, toC = C.structure()
sage: fromC(c)
sqrt2 - sqrt3

sage: toC(a)
1/2*c^3 - 9/2*c

sage: toC(b)
1/2*c^3 - 11/2*c
```

This tells us that the single generator of the flattened tower, `c`, is equal to $\sqrt{2}-\sqrt{3}$, and further, each of $\sqrt{2}$ and $\sqrt{3}$ can be expressed as polynomial functions of `c`. With these connections, you might want to compute the final two expressions in `c` by hand, and appreciate the work Sage does to determine these for us. This computation is an example of the conclusion of the upcoming Theorem 23.8.

Many number field methods have both relative and absolute versions, and we will also find it more convenient to work in a tower or a flattened version, thus the isomorphisms between the two can be invaluable for translating both questions and answers. As a vector space over $\mathbb{Q}$, or over another number field, number fields that are finite extensions have a dimension, called the degree. These are easy to get from Sage, though for a relative field, we need to be more precise about which degree we desire.

```
sage: B.degree()
2

sage: A.absolute_degree()
4

sage: A.relative_degree()
2
```

21.1.3 Splitting Fields

Here is a concrete example of how to use Sage to construct a splitting field of a polynomial. Consider $p(x) = x^4 + x^2 - 1$. We first build a number field with a single root, and then factor the polynomial over this new, larger, field.

```
sage: x = polygen(QQ, 'x')
sage: p = x^4 + x^2 - 1
sage: p.parent()
Univariate Polynomial Ring in x over Rational Field

sage: p.is_irreducible()
True

sage: M.<a> = NumberField(p, 'a')
sage: y = polygen(M, 'y')
sage: p = p.subs(x = y); p
y^4 + y^2 - 1

sage: p.parent()
Univariate Polynomial Ring in y over Number Field in a with
defining polynomial x^4 + x^2 - 1

sage: p.factor()
(y - a) * (y + a) * (y^2 + a^2 + 1)
```

So our polynomial factors partially into two linear factors and a quadratic factor. But notice that the quadratic factor has a coefficient that is irrational, $a^2 + 1$, so the quadratic factor properly belongs in the polynomial ring over `M`.

```
sage: q = y^2 + a^2 + 1
sage: N.<b> = NumberField(q, 'b')
sage: z = polygen(N, 'z')
sage: p = p.subs(y = z); p
z^4 + z^2 - 1

sage: p.parent()
Univariate Polynomial Ring in z over Number Field in b with
defining polynomial y^2 + a^2 + 1 over its base field

sage: p.factor()
(z + b) * (z + a) * (z - a) * (z - b)
```

This factorization is a bit unsettling, since `p` is clearly a polynomial in `z`, but is being factored as a polynomial in `x`. However, it clearly shows `p` factored into four linear factors with the roots of the polynomial in terms of the generators `a` and `b`. We can get another factorization by converting `N` to an absolute number field and factoring there. We need to recreate the polynomial over `N`, since a substitution will carry coefficients from the wrong ring.

```
sage: P.<c> = N.absolute_field()
sage: w = polygen(P, 'w')
sage: p = w^4 + w^2- 1
sage: p.factor()
(w - 7/18966*c^7 + 110/9483*c^5 +  923/9483*c^3 +  3001/6322*c) *
(w - 7/37932*c^7 +  55/9483*c^5 + 923/18966*c^3 - 3321/12644*c) *
(w + 7/37932*c^7 -  55/9483*c^5 - 923/18966*c^3 + 3321/12644*c) *
(w + 7/18966*c^7 - 110/9483*c^5 -  923/9483*c^3 -  3001/6322*c)
```

This is an improvement, in that the variable is `w` and the roots of the polynomial are expressions in terms of the single generator `c`. Thus `P` (or `N`) is a splitting field for $p(x) = x^4 + x^2 - 1$. The roots are not really as bad as they appear — lets convert them back to the relative number field.

First we want to rewrite a single factor (the first) in the form $(w - r)$ to identify the root with the correct signs.

```
(w -  7/18966*c^7 + 110/9483*c^5 + 923/9483*c^3 + 3001/6322*c) =
(w - (7/18966*c^7 - 110/9483*c^5 - 923/9483*c^3 - 3001/6322*c))
```

With the converting isomorphisms, we can recognize the roots for what they are.

```
sage: fromP, toP = P.structure()
sage: fromP(7/18966*c^7 - 110/9483*c^5 - 923/9483*c^3 - 3001/6322*c)
-b
```

So the rather complicated expression in `c` is just the negative of the root we adjoined in the second step of constructing the tower of number fields. It would be a good exercise to see what happens to the other three roots (being careful to get the signs right on each root). This is a good opportunity to illustrate Theorem 21.7.

```
sage: M.degree()
4

sage: N.relative_degree()
2

sage: P.degree()
8

sage: M.degree()*N.relative_degree() == P.degree()
True
```

21.1.4 Algebraic Numbers

Corollary 21.12 says that the set of *all* algebraic numbers forms a field. This field is implemented in Sage as `QQbar`. This allows for finding roots of polynomials as exact quantities which display as inexact numbers.

```
sage: x = polygen(QQ, 'x')
sage: p = x^4 + x^2 - 1
sage: r = p.roots(ring=QQbar); r
[(-0.7861513777574233?,  1), (0.7861513777574233?,  1),
 (-1.272019649514069?*I, 1), (1.272019649514069?*I, 1)]
```

So we asked for the roots of a polynomial over the rationals, but requested any root that may lie outside the rationals and within the field of algebraic numbers. Since the field of algebraic numbers contains all such roots, we get a full four roots of the fourth-degree polynomial. These roots are computed to lie within an interval and the question mark indicates that the preceding digits are accurate. (The integers paired with each roo are the multiplicities of each root. Use the keyword `multiplicities=False` to turn them off.) Lets take a look under the hood and see how Sage manages the field of algebraic numbers.

```
sage: r1 = r[0][0]; r1
-0.7861513777574233?

sage: r1.as_number_field_element()
(Number Field in a with defining polynomial y^4 - y^2 - 1,
 a^3 - a,
 Ring morphism:
   From: Number Field in a with defining polynomial y^4 - y^2 - 1
   To:   Algebraic Real Field
   Defn: a |--> -1.272019649514069?)
```

Three items are associated with this initial root. First is a number field, with generator `a` and a defining polynomial similar to the polynomial we the finding roots of, but not identical. Second is an expression in the generator `a` which is the actual

root. Finally, there is a ring homomorphism from the number field to the "Algebraic Real Field", `AA`, the subfield of `QQbar` with just real elements, which associates the generator `a` with the number `-1.272019649514069?`. Let's verify that the root given is really a root, in two ways.

```
sage: r1^4 + r1^2 - 1
0

sage: N, rexact, homomorphism = r1.as_number_field_element()
sage: (rexact)^4 + rexact^2 - 1
0
```

21.1.5 Geometric Constructions

Sage can do a lot of things, but it is not yet able to lay out lines with a straightedge and compass. However, we can very quickly determine that trisecting a 60 degree angle is impossible. We adjoin the cosine of a 20 degree angle (in radians) to the rationals, determine the degree of the extension, and check that it is not an integer power of 2. In one line. Sweet.

```
sage: log(QQ[cos(pi/9)].degree(), 2) in ZZ
False
```

21.2 Exercises

1 Create the polynomial $p(x) = x^5 + 2x^4 + 1$ over $\mathbb{Z}_3$. Verify that it does not have any linear factors by evaluating $p(x)$ with each element of $\mathbb{Z}_3$, and then check that $p(x)$ is irreducible.

Create a finite field of order 3^5 with the `FiniteField()` command, but include the `modulus` keyword set to the polynomial $p(x)$ to override the default choice.

Recreate $p(x)$ as a polynomial over this field and check each of the $3^5 = 243$ elements of the field to see if they are roots of the polynomial. Finally request a factorization of $p(x)$ over the field.

2 This problem continues the previous one. Build the ring of polynomials over $\mathbb{Z}_3$ and within this ring use $p(x)$ to generate a principal ideal. Finally construct the quotient of the polynomial ring by the ideal. Since the polynomial is irreducible, this quotient ring is a field, and by Proposition 21.4 this quotient ring is isomorphic to the number field in the previous problem.

Create five roots of the polynomial $p(x)$ within this quotient ring, as expressions in the generator of the quotient ring (which is technically a coset). Use Sage to verify that they are indeed roots. This demonstrates using a quotient ring to create a splitting field for an irreducible polynomial over a finite field.

3 In the section of this chapter relying on techniques from linear algebra, the text proves that every finite extension is an algebraic extension. This exercise will help you understand this proof.

The polynomial $r(x) = x^4 + 2x + 2$ is irreducible over the rationals (Eisenstein's criterion with prime $p = 2$). Create a number field that contains a root of $r(x)$. By Theorem 21.6, and the remark following, every element of this finite field extension is an algebraic number, and hence satisfies some polynomial over the base field (it is this polynomial that Sage will produce with the `.minpoly()` method). This exercise will show how we can use just linear algebra to determine this minimal polynomial.

Suppose that `a` is the generator of the number field you just created with $r(x)$. Then we will determine the minimal polynomial of `t = 3a + 1` using just linear algebra. According to the proof, the first five powers of `t` (start counting from zero) will be linearly dependent. Why? So a nontrivial relation of linear dependence on these powers will provide the coefficients of a polynomial with `t` as a root. Compute these five powers, form the right linear system to determine the coefficients of the minimal polynomial, solve the system and suitably interpret its solutions.

Hints: The `vector()` and `matrix()` commands will create vectors and matrices, and the `.solve_right()` method for matrices can be used to find solutions. Given an element of the number field, which will necessarily be a polynomial in the generator `a`, the `vector()` method on the element will provide the coefficients of this polynomial in a list.

4 Construct the splitting field of $s(x) = x^4 + x^2 + 1$ and find a factorization of $s(x)$ over this field into linear factors.

5 Form the number field, K, which contains a root of the irreducible polynomial $q(x) = x^3 + 3x^2 + 3x - 2$. Verify that $q(x)$ factors, but does not split, over K. With K now as the base field, form an extension of K where the quadratic factor of $q(x)$ has a root. Call this second extension of the tower, L.

Use `M.<c> = L.absolute_field()` to form the flattened tower that is the absolute number field `M`. Find the defining polynomial of `M` with the `.polynomial()` method. From this polynomial, which must have the generator `c` as a root, you should be able to use elementary algebra to write the generator as a fairly simple expression.

M should be the splitting field of $q(x)$. To see this, start over, and build from scratch a new number field, P, using the simple expression for `c` that you just found. Then factor the original polynomial $q(x)$ (with rational coefficients) over P, to see that the polynomial really does split. Using this factorization, and your simple expression for `c` write simplified expressions for the three roots of $q(x)$.

Chapter 22

Finite Fields

22.1 Discussion

You have noticed in this chapter that finite fields have a great deal of structure. We have also seen finite fields in Sage regularly as examples of rings and fields. Now we can combine the two, mostly using commands we already know, plus a few new ones.

22.1.1 Creating Finite Fields

By Theorem 22.5 we know that all finite fields of a given order are isomorphic and that possible orders are limited to powers of primes. We can use the `FiniteField()` command, as before, or a shorter equivalent is `GF()`. Optionally, we can specify an irreducible polynomial for the contruction of the field. We can view this polynomial as the generator of the principal ideal of a polynomial ring, or we can view it as a "re-writing" rule for powers of the field's generator that allow us to multiply elements and reformulate them as linear combinations of lesser powers.

Absent providing an irreducible polynomial, Sage will use a Conway polynomial. You can determine these with the `conway_polynomial()` command, or just build a finite field and request the defining polynomial with the `.polynomial()` method.

```
sage: F.<a> = GF(7^15); F
Finite Field in a of size 7^15

sage: F.polynomial()
a^15 + 5*a^6 + 6*a^5 + 6*a^4 + 4*a^3 + a^2 + 2*a + 4

sage: a^15 + 5*a^6 + 6*a^5 + 6*a^4 + 4*a^3 + a^2 + 2*a + 4
0

sage: conway_polynomial(7, 15)
x^15 + 5*x^6 + 6*x^5 + 6*x^4 + 4*x^3 + x^2 + 2*x + 4

sage: y = polygen(Integers(7), 'y')
```

Just to be more readable, we coerce a list of coefficients into the set of polynomials (obtained with the `.parent()` method on a simple polynomial) to define a polynomial.

```
sage: y = polygen(Integers(7), 'y')
sage: P = y.parent()
sage: p = P([4, 5, 2, 6, 3, 3, 6, 2, 1, 1, 2, 5, 6, 3, 5, 1]); p
y^15 + 5*y^14 + 3*y^13 + 6*y^12 + 5*y^11 + 2*y^10 + y^9 +
y^8 + 2*y^7 + 6*y^6 + 3*y^5 + 3*y^4 + 6*y^3 + 2*y^2 + 5*y + 4

sage: p.is_irreducible()
True

sage: T.<b> = GF(7^15, modulus=p); T
Finite Field in b of size 7^15
```

One useful command we have not described is the `.log()` method for elements of a finite field. Since we now know that the multiplicative group of nonzero elements is cyclic, we can express every element as a power of the generator. The `log` method will return that power.

Usually we will want to use the generator as the base of a logaithm computation in a finite field. However, other bases may be used, wih the understanding that if the base is not a generator, then the logarithm may note exist (i.e. there may not be a solution to the relevant equation).

```
sage: F.<a> = GF(5^4)
sage: a^458
3*a^3 + 2*a^2 + a + 3

sage: (3*a^3 + 2*a^2 + a + 3).log(a)
458

sage: exponent = (3*a^3 + 2*a^2 + a + 3).log(2*a^3 + 4*a^2 + 4*a)
sage: exponent
211

sage: (2*a^3 + 4*a^2 + 4*a)^exponent == 3*a^3 + 2*a^2 + a + 3
True

sage: (3*a^3 + 2*a^2 + a + 3).log(a^2 + 4*a + 4)
Traceback (most recent call last):
...
ValueError: No discrete log of 3*a^3 + 2*a^2 + a + 3 found
to base a^2 + 4*a + 4
```

Since we already know many Sage commands, there is nothing else to introduce before we can work profitably with finite fields. The exercises explore the ways we can examine and exploit the structure of finite fields in Sage.

22.2 Exercises

1 Create a finite field of order 5^2 and then factor $p(x) = x^{25} - x$ over this field. Comment on what is interesting about this result and why it is not a surprise.

2 Corollary 22.8 says that the nonzero elements of a finite field are a cyclic group under multiplication. The generator used in Sage is also a generator of this multiplicative group. To see this, create a finite field of order 2^7. Create two lists of the elements of the field: first, use the `.list()` method, then use a list comprehension to generate the proper powers of the generator you specified when you created the field.

The second list should be the whole field, but will be missing zero. Create the zero element of the field (perhaps by coercing 0 into the field) and `.append()` it to the list of powers. Apply the `sorted()` command to each list and then test the lists for equality.

3 While subfields of a finite field are completely classified by Theorem 22.6, unfortunately they are not easily created in Sage. In this exercise we will create a subfield of a finite field. Since the group of nonzero elements in a finite field is cyclic, the nonzero elements of a subfield will form a subgroup of the cyclic group, and necessarily will be cyclic.

Create a finite field of order 3^6. Theory says there is a subfield of order 3^2. Determine a generator of multiplicative order 8 for the nonzero elements of this subfield, and construct these 8 elements. Add in the field's zero element to this set. It is fairly obvious that this set of 9 elements is closed under multiplication. Write a single statement that checks if this set is also closed under addition by considering all possible sums of elements from the set.

4 This problem investigates the "separableness" of $\mathbb{Q}(\sqrt{3}, \sqrt{7})$. YOu can create this number field quickly with the `NumberFieldTower` constructor, along with the polynomials $x^2 - 3$ and $x^2 - 7$. Flatten the tower with the `.absolute_field()` method and use the `.structure()` method to retrieve mappings between the tower and the flattened version. Name the tower `N` and use `a` and `b` as generators and name the flattened version `L` with `c` as a generator.

Create a nontrivial ("random") element of `L` using as many powers of `c` as possible (check the degree of `L` to see how many linearly independent powers there are). Request from Sage the minimum polynomial of your random element, thus ensuring the element is a root. Construct the minimum polynomial as a polynomial over `N`, the field tower, and find its factorization. Your factorization should have only linear

factors. Each root should be an expression in `a` and `b`, so convert each root into an expression with mathematical notation involving $\sqrt{3}$ and $\sqrt{7}$. Use one of the mappings to verify that one of the roots is indeed the original random element.

Create a few more random elements, and find a factorization (in N or in L). For a field to be separable, every element of the field should be a root of *some* separable polynomial. The minimal polynomial is a good polynomial to test. (Why?) Based on the evidence, does it appear that $\mathbb{Q}(\sqrt{3}, \sqrt{7})$ is a separable extension?

5 Exercise 21 in this chapter describes the "Frobenius Map," an automorphism of a finite field. If `F` is a finite field in Sage, then `End(F)` will create the automorphism group of F, the set of all bijective mappings betwwen the field and itself.
(a) Work Exercise 21 to gain an understanding of how and why the Frobenius mapping is a field automorphism. (Don't include any of this in yor answer to this question, but understand that the following will be much easier if you do this problem first.)
(b) For some small, but not trivial, finite fields locate the Frobenius map in the automorphism group. Small might mean $p = 2, 3, 5, 7$ and $3 \leq n \leq 10$, with n prime vs. composite.
(c) Once you have located the Frobenius map, describe the other automorphisms. In other words, with a bit of investigation, you should find a description of the automorphisms which will allow you to accurately predict the entire automorphism group for a finite field you have not already explored. (Hint: the automorphism group is a group. What if you "do the operation" between the Frobenius map and itself? Just what is the operation? Try using Sage's multiplicative notation with the elements of the automorphism group.)
(d) What is the "structure" of the automorphism group? What special status does the Frobenius map have in this group?
(e) For any field, the subfield known as the fixed field is an important construction, and will be especially important in the next chapter. Given an automorphism τ of a field E, the fixed field of τ in E is $K = \{b \in E \mid \tau(b) = b\}$. For each automorphism of $E = GF(3^6)$ identify the fixed field of the automorphism. Since we understand the structure of subfields of a finite field, it is enough to just determine the order of the fixed field to be able to identify the subfield precisely.

Chapter 23

Galois Theory

23.1 Discussion

Again, our competence at examining fields with Sage will allow us to study the main concepts of Galois Theory easily. We will thoroughly examine Example 7 carefully using our computational tools.

23.1.1 Galois Groups

We will repeat Example 7 and analyze carefully the splitting field of the polynomial $p(x) = x^4 - 2$. We begin with an initial field extension containing at least one root.

```
sage: x = polygen(QQ, 'x')
sage: N.<a> = NumberField(x^4 - 2); N
Number Field in a with defining polynomial x^4 - 2
```

The `.galois_closure()` method will create an extension containing all of the roots of the defining polynomial of a number field.

```
sage: L.<b> = N.galois_closure(); L
Number Field in b with defining polynomial x^8 + 28*x^4 + 2500

sage: L.degree()
8

sage: y = polygen(L, 'y')
sage: (y^4 - 2).factor()
(y - 1/120*b^5 -  19/60*b) *
(y - 1/240*b^5 + 41/120*b) *
(y + 1/240*b^5 - 41/120*b) *
(y + 1/120*b^5 +  19/60*b)
```

From the factorization, it is clear that `L` is the splitting field of the polynomial, even if the factorization is not pretty. It is easy to then obtain the Galois group of this field extension.

```
sage: G = L.galois_group(); G
Galois group of Number Field in b with
defining polynomial x^8 + 28*x^4 + 2500
```

We can examine this group, and identify it. Notice that since the field is a degree 8 extension, the group is described as a permutation group on 8 symbols. (It is just a coincidence that the group has 8 elements.) With a paucity of nonabelian groups of order 8, it is not hard to guess the nature of the group.

```
sage: G.is_abelian()
False

sage: G.order()
8

sage: G.list()
[(), (1,2,8,7)(3,4,6,5),
(1,3)(2,5)(4,7)(6,8), (1,4)(2,3)(5,8)(6,7),
(1,5)(2,6)(3,7)(4,8), (1,6)(2,4)(3,8)(5,7),
(1,7,8,2)(3,5,6,4), (1,8)(2,7)(3,6)(4,5)]

sage: G.is_isomorphic(DihedralGroup(4))
True
```

That's it. But maybe not very satisfying. Let's dig deeper for more understanding. We will start over and create the splitting field of $p(x) = x^4 - 2$ again, but the primary difference is that we will make the roots extremely obvious so we can work more carefully with the Galois group and the fixed fields. Along the way, we will see another example of linear algebra enabling certain computations. The following construction should be familiar by now.

```
sage: x = polygen(QQ, 'x')
sage: p = x^4 - 2
sage: N.<a> = NumberField(p); N
Number Field in a with defining polynomial x^4 - 2

sage: y = polygen(N, 'y')
sage: p = p.subs(x=y)
sage: p.factor()
(y - a) * (y + a) * (y^2 + a^2)

sage: M.<b> = NumberField(y^2 + a^2); M
Number Field in b with defining polynomial y^2 + a^2 over
its base field

sage: z = polygen(M, 'z')
sage: (z^4 - 2).factor()
(z - b) * (z - a) * (z + a) * (z + b)
```

The important thing to notice here is that we have arranged the splitting field so that the four roots, `a, -a, b, -b`, are very simple functions of the generators. In more traditional notation, `a` is $2^{\frac{1}{4}} = \sqrt[4]{2}$, and `b` is $2^{\frac{1}{4}}i = \sqrt[4]{2}i$.

We will find it easier to compute in the flattened tower, a now familiar construction.

```
sage: L.<c> = M.absolute_field(); L
Number Field in c with defining polynomial x^8 + 28*x^4 + 2500

sage: fromL, toL = L.structure()
```

We can return to our original polynomial (over the rationals), and ask for its roots in the flattened tower, custom-designed to contain these roots.

```
sage: roots = p.roots(ring=L, multiplicities=False); roots
[1/120*c^5 +  19/60*c,
 1/240*c^5 - 41/120*c,
-1/240*c^5 + 41/120*c,
-1/120*c^5 -  19/60*c]
```

Hmmm. Do those look right? If you look back at the factorization obtained in the field constructed with the `.galois_closure()` method, then they look right. But we can do better.

```
sage: [fromL(r) for r in roots]
[b, a, -a, -b]
```

Yes, those are the roots. The `End()` command will create the group of automorphisms of the field `L`.

```
sage: G = End(L); G
Automorphism group of Number Field in c with
defining polynomial x^8 + 28*x^4 + 2500
```

We can check that each of these automorphisms fixes the rational numbers elementwise. If a field homomorphism fixes 1, then it will fix the integers, and thus fix all fractions of integers.

```
sage: [tau(1) for tau in G]
[1, 1, 1, 1, 1, 1, 1, 1]
```

So each element of `G` fixes the rationals elementwise and thus `G` is the Galois group of the splitting field `L` over the rationals.

Proposition 23.3 is fundamental. It says every automorphism in the Galois group of a field extension creates a permutation of the roots of a polynomial with coefficients in the base field. We have all of those ingredients here. So we will evaluate each automorphism of the Galois group at each of the four roots of our polynomial, which in each case should be another root. (We use the `Sequence()` constructor just to get nicely-aligned output.)

```
sage: Sequence([[fromL(tau(r)) for r in roots] for tau in G], cr=True)
[
[b, a, -a, -b],
[-b, -a, a, b],
[a, -b, b, -a],
[b, -a, a, -b],
[-a, -b, b, a],
[a, b, -b, -a],
[-b, a, -a, b],
[-a, b, -b, a]
]
```

Each row of the output is a list of the roots, but permuted, and so corresponds to a permutation of four objects (the roots). For example, the second row shows the second automorphism interchanging `a` with `-a`, and `b` with `-b`. (notice that the first row is the result of the identity automorphism, so we can mentally comine the first row with any other row to imagine a "two-row" form of a permutation.) We can number the roots, 1 through 4, and create each permutation as an element of S_4. It is overkill, but we can then build the permutation group by letting *all* of these elements generate a group.

```
sage: S4 = SymmetricGroup(4)
sage: elements = [S4([1, 2, 3, 4]),
...               S4([4, 3, 2, 1]),
...               S4([2, 4, 1, 3]),
...               S4([1, 3, 2, 4]),
...               S4([3, 4, 1, 2]),
...               S4([2, 1, 4, 3]),
...               S4([4, 2, 3, 1]),
...               S4([3, 1, 4, 2])]
sage: elements
[(), (1,4)(2,3), (1,2,4,3), (2,3), (1,3)(2,4),
 (1,2)(3,4), (1,4), (1,3,4,2)]

sage: P = S4.subgroup(elements)
sage: P.is_isomorphic(DihedralGroup(4))
True
```

Notice that we now have built an isomorphism from the Galois group to a group of permutations *using just four symbols*, rather than the eight used previously.

23.1.2 Fixed Fields

In a previous exercise, we computed the fixed fields of single field automorphisms for finite fields. This was "easy" in the sense that we could just test every element of the field to see if it was fixed, since the field was finite. Now we have an infinite field

extension. How are we going to determine which elements are fixed by individual automorphisms, or subgroups of automorphisms?

The answer is to use the vector space structure of the flattened tower. As a degree 8 extension of the rationals, the first 8 powers of the primitive element `c` form a basis when the field is viewed as a vector space with the rationals as the scalars. It is sufficient to know how each field automorphism behaves on this basis to fully specify the definition of the automorphism. To wit,

$$\begin{aligned}
\tau(x) &= \tau\left(\sum_{i=0}^{7} q_i c^i\right) && q_i \in \mathbb{Q} \\
&= \sum_{i=0}^{7} \tau(q_i)\tau(c^i) && \tau \text{ is a field automorphism} \\
&= \sum_{i=0}^{7} q_i \tau(c^i) && \text{rationals are fixed}
\end{aligned}$$

So we can compute the value of a field automorphism at any linear combination of powers of the primitive element as a linear combination of the values of the field automorphism at just the powers of the primitive element. This is known as the "power basis", which we can obtain simply with the `.power_basis()` method. We will begin with an example of how we can use this basis. We will illustrate with the fourth automorphism of the Galois group. Notice that the `.vector()` method is a convenience that strips a linear combination of the powers of `c` into a vector of just the coefficients. (Notice too that τ is totally defined by the value of $\tau(c)$, since as a field automorphism $\tau(c^k) = (\tau(c))^k$. However, we still need to work with the entire power basis to exploit the vector space structure.)

```
sage: basis = L.power_basis(); basis
[1, c, c^2, c^3, c^4, c^5, c^6, c^7]

sage: tau = G[3]
sage: z = 4 + 5*c+ 6*c^3-7*c^6
sage: tz = tau(4 + 5*c+ 6*c^3-7*c^6); tz
11/250*c^7 - 98/25*c^6 + 1/12*c^5 + 779/125*c^3 +
6006/25*c^2 - 11/6*c + 4

sage: tz.vector()
(4, -11/6, 6006/25, 779/125, 0, 1/12, -98/25, 11/250)

sage: tau_matrix = column_matrix([tau(be).vector() for be in basis])
sage: tau_matrix
[  1       0       0        0  -28       0        0          0]
[  0  -11/30       0        0    0  779/15        0          0]
[  0       0  -14/25        0    0       0  -858/25          0]
[  0       0       0  779/750    0       0        0  -4031/375]
```

```
[  0       0       0        0   -1       0        0           0]
[  0    1/60       0        0    0   11/30        0           0]
[  0       0   -1/50        0    0       0    14/25           0]
[  0       0       0  11/1500    0       0        0   -779/750]
```

```
sage: tau_matrix*z.vector()
(4, -11/6, 6006/25, 779/125, 0, 1/12, -98/25, 11/250)

sage: tau_matrix*(z.vector()) == (tau(z)).vector()
True
```

The last line expresses the fact that `tau_matrix` is a matrix representation of the field automorphism, viewed as a linear transformation of the vector space structure. As a representation of an invertible field homomorphism, the matrix is invertible, and as an order 2 permutation of the roots, the inverse of the matrix is itself. But these facts are just verifications that we have the right thing, we are interested in other properties.

To construct fixed fields, we want to find elements fixed by automorphisms. Continuing with tau from above, we seek elements `z` (written as vectors) such that `tau_matrix*z=z`. These are eigenvectors for the eigenvalue 1, or elements of the null space of `(tau_matrix - I)` (null spaces are obtained with `.right_kernel()` in Sage).

```
sage: K = (tau_matrix-identity_matrix(8)).right_kernel(); K
Vector space of degree 8 and dimension 4 over Rational Field
Basis matrix:
[    1     0     0     0     0     0     0     0]
[    0     1     0     0     0  1/38     0     0]
[    0     0     1     0     0     0 -1/22     0]
[    0     0     0     1     0     0     0 1/278]
```

Each row of the basis matrix is a vector representing an element of the field, specifically `1`, `c + (1/38)*c^5`, `c^2 - (1/22)*c^6`, `c^3 + (1/278)*c^7`. Let's take a closer look at these fixed elements, in terms we recognize.

```
sage: fromL(1)
1

sage: fromL(c + (1/38)*c^5)
60/19*b

sage: fromL(c^2 - (1/22)*c^6)
150/11*a^2

sage: fromL(c^3 + (1/278)*c^7)
1500/139*a^2*b
```

Any element fixed by `tau` will be a linear combination of these four elements. We can ignore any rational multiples present, the first element is just saying the rationals are fixed, and the last element is just a product of the middle two. So fundamentally `tau` is fixing rationals, `b` (which is $\sqrt[4]{2}i$) and `a^2` (which is $\sqrt{2}$). Furthermore, `b^2 = -a^2` (check below), so we can create any fixed element by just adjoining $\sqrt[4]{2}i$ to the rationals. So the elements fixed by `tau` are $\mathbb{Q}(\sqrt[4]{2}i)$.

```
sage: a^2 + b^2
0
```

23.1.3 Galois Correspondence

The entire subfield structure of our splitting field is determined by the subgroup structure of the Galois group (Theorem 23.15), which is isomorphic to a group we know well. What are the subgroups of our Galois group, expressed as a permutation group? (The `Sequence()` constructor with the `cr=True` option will format the output nicely.)

```
sage: sg = P.subgroups(); sg
[Permutation Group with generators [()],
 Permutation Group with generators [(2,3)],
 Permutation Group with generators [(1,4)],
 Permutation Group with generators [(1,4)(2,3)],
 Permutation Group with generators [(1,2)(3,4)],
 Permutation Group with generators [(1,3)(2,4)],
 Permutation Group with generators [(2,3), (1,4)],
 Permutation Group with generators [(1,3)(2,4), (1,4)(2,3)],
 Permutation Group with generators [(1,3,4,2), (1,4)(2,3)],
 Permutation Group with generators [(2,3), (1,3)(2,4), (1,4)(2,3)]]

sage: [H.order() for H in sg]
[1, 2, 2, 2, 2, 2, 4, 4, 4, 8]
```

`tau` above is the fourth element of the automorphism group, and the fourth permutation in `elements` is the permutation `(2,3)`, the generator (of order 2) for the second subgroup. So as the only nontrivial element of this subgroup, we know that the corresponding fixed field is $\mathbb{Q}(\sqrt[4]{2}i)$.

Let's analyze another subgroup of order 2, without all the explanation, and starting with the subgroup. The sixth subgroup is generated by the fifth automorphism, so let's determine the elements that are fixed.

```
sage: tau = G[4]
sage: tau_matrix = column_matrix([tau(be).vector() for be in basis])
sage: (tau_matrix-identity_matrix(8)).right_kernel()
Vector space of degree 8 and dimension 4 over Rational Field
Basis matrix:
```

```
[     1      0      0      0      0      0      0      0]
[     0      1      0      0      0  1/158      0      0]
[     0      0      1      0      0      0   1/78      0]
[     0      0      0      1      0      0      0 13/614]

sage: fromL(tau(1))
1

sage: fromL(tau(c+(1/158)*c^5))
120/79*b - 120/79*a

sage: fromL(tau(c^2+(1/78)*c^6))
-200/39*a*b

sage: fromL(tau(c^3+(13/614)*c^7))
3000/307*a^2*b + 3000/307*a^3
```

The first element indicates that the rationals are fixed (we knew that). Scaling the second element gives `b - a` as a fixed element. Scaling the third and fourth fixed elements, we recognize that they can be obtained from powers of `b - a`.

```
sage: (b-a)^2
-2*a*b

sage: (b-a)^3
2*a^2*b + 2*a^3
```

So the fixed field of this subgroup can be formed by adjoining `b - a` to the rationals, which in mathematical notation is $\sqrt[4]{2}i - \sqrt[4]{2} = (1-i)\sqrt[4]{2}$, so the fixed field is $\mathbb{Q}(\sqrt[4]{2}i - \sqrt[4]{2} = (1-i)\sqrt[4]{2})$

We can create this fixed field, though as created here it is not strictly a subfield of `L`. We will use an expression for `b - a` that is a linear combination of powers of `c`.

```
sage: subinfo = L.subfield((79/120)*(c+(1/158)*c^5)); subinfo
(Number Field in c0 with defining polynomial x^4 + 8, Ring morphism:
  From: Number Field in c0 with defining polynomial x^4 + 8
  To:   Number Field in c with defining polynomial x^8 + 28*x^4 + 2500
  Defn: c0 |--> 1/240*c^5 + 79/120*c)
```

The `.subfield()` method returns a pair. The first item is a new number field, isomorphic to a subfield of `L`. The second item is an injective mapping from the new number field into `L`. In this case, the image of the primitive element `c0` is the element we have specified as the generator of the subfield. The primitive element of the new field will satisfy the defining polynomial $x^4 + 8$ — you can check that $(1-i)\sqrt[4]{2}$ is indeed a root of the polynomial $x^4 + 8$.

There are five subgroups of order 2, we have found fixed fields for two of them. The other three are similar, so it would be a good exercise to work through them. Our

automorphism group has three subgroups of order 4, and at least one of each possible type (cyclic versus non-cyclic). Fixed fields of larger subgroups require that we find elements fixed by all of the automorphisms in the subgroup. (We were conveniently ignoring the identity automorphism above.) This will require more computation, but will restrict the possibilities (smaller fields) to where it will be easier to deduce a primitive element for each field.

The seventh subgroup is generated by two elements of order 2 and is composed entirely of elements of order 2 (except the identity), so is isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_2$. The permutations correspond to automorphisms number 0, 1, 3 and 6. To determine the elements fixed by *all four* automorphisms, we will build the kernel for each one and as we go form the *intersection* of all four kernels. We will work via a loop over the four automorphisms.

```
sage: V = QQ^8
sage: for tau in [G[0], G[1], G[3], G[6]]:
...     tau_matrix = column_matrix([tau(be).vector() for be in basis])
...     K = (tau_matrix-identity_matrix(8)).right_kernel()
...     V = V.intersection(K)
sage: V
Vector space of degree 8 and dimension 2 over Rational Field
Basis matrix:
[    1     0     0     0     0     0     0     0]
[    0     0     1     0     0     0 -1/22     0]
```

Outside the rationals, there is a single fixed element.

```
sage: fromL(tau(c^2 - (1/22)*c^6))
150/11*a^2
```

Removing a scalar multiple, our primitive element is `a^2`, which mathematically is $\sqrt{2}$, so the fixed field is $\mathbb{Q}(\sqrt{2})$. Again, we can build this fixed field, but ignore the mapping.

```
sage: F, mapping = L.subfield((11/150)*(c^2 - (1/22)*c^6))
sage: F
Number Field in c0 with defining polynomial x^2 - 2
```

One more subgroup. The penultimate subgroup has a permutation of order 4 as a generator, so is a cyclic group of order 4. The individual permutations of the subgroup correspond to automorphisms 0, 1, 2, 7.

```
sage: V = QQ^8
sage: for tau in [G[0], G[1], G[2], G[7]]:
...     tau_matrix = column_matrix([tau(be).vector() for be in basis])
...     K = (tau_matrix-identity_matrix(8)).right_kernel()
...     V = V.intersection(K)
sage: V
```

```
Vector space of degree 8 and dimension 2 over Rational Field
Basis matrix:
[1 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0]
```

So we compute the primitive element.

```
sage: fromL(tau(c^4))
-24*a^3*b - 14
```

Since rationals are fixed, we can remove the -14 and the multiple and take `a^3*b` as the primitive element. Mathematically, this is $2i$, so we might as well use just i as the primitive element and the fixed field is $\mathbb{Q}(i)$. We can then build the fixed field (and ignore the mapping also returned).

```
sage: F, mapping = L.subfield((c^4+14)/-48)
sage: F
Number Field in c0 with defining polynomial x^2 + 1
```

There is one more subgroup of order 4, which we will leave as an exercise to analyze. There are also two trivial subgroups (the identity and the full group) which are not very interesting or surprising.

If the above seems like too much work, you can always just have Sage do it all with the `.subfields()` method.

```
sage: L.subfields()
[
(Number Field in c0 with defining polynomial x,
 Ring morphism:
   From: Number Field in c0 with defining polynomial x
   To:   Number Field in c with defining polynomial x^8 + 28*x^4 + 2500
   Defn: 0 |--> 0,
 None),
(Number Field in c1 with defining polynomial x^2 + 112*x + 40000,
 Ring morphism:
   From: Number Field in c1 with defining polynomial x^2 + 112*x + 40000
   To:   Number Field in c with defining polynomial x^8 + 28*x^4 + 2500
   Defn: c1 |--> 4*c^4,
 None),
(Number Field in c2 with defining polynomial x^2 + 512,
 Ring morphism:
   From: Number Field in c2 with defining polynomial x^2 + 512
   To:   Number Field in c with defining polynomial x^8 + 28*x^4 + 2500
   Defn: c2 |--> 1/25*c^6 + 78/25*c^2,
 None),
(Number Field in c3 with defining polynomial x^2 - 288,
 Ring morphism:
```

```
    From: Number Field in c3 with defining polynomial x^2 - 288
    To:   Number Field in c with defining polynomial x^8 + 28*x^4 + 2500
    Defn: c3 |--> -1/25*c^6 + 22/25*c^2,
  None),
 (Number Field in c4 with defining polynomial x^4 + 112*x^2 + 40000,
  Ring morphism:
    From: Number Field in c4 with defining polynomial x^4 + 112*x^2 + 40000
    To:   Number Field in c with defining polynomial x^8 + 28*x^4 + 2500
    Defn: c4 |--> 2*c^2,
  None),
 (Number Field in c5 with defining polynomial x^4 + 648,
  Ring morphism:
    From: Number Field in c5 with defining polynomial x^4 + 648
    To:   Number Field in c with defining polynomial x^8 + 28*x^4 + 2500
    Defn: c5 |--> 1/80*c^5 + 79/40*c,
  None),
 (Number Field in c6 with defining polynomial x^4 + 8,
 Ring morphism:
   From: Number Field in c6 with defining polynomial x^4 + 8
   To:   Number Field in c with defining polynomial x^8 + 28*x^4 + 2500
   Defn: c6 |--> -1/80*c^5 + 1/40*c,
   None),
 (Number Field in c7 with defining polynomial x^4 - 512,
  Ring morphism:
    From: Number Field in c7 with defining polynomial x^4 - 512
    To:   Number Field in c with defining polynomial x^8 + 28*x^4 + 2500
    Defn: c7 |--> -1/60*c^5 + 41/30*c,
  None),
 (Number Field in c8 with defining polynomial x^4 - 32,
  Ring morphism:
    From: Number Field in c8 with defining polynomial x^4 - 32
    To:   Number Field in c with defining polynomial x^8 + 28*x^4 + 2500
    Defn: c8 |--> 1/60*c^5 + 19/30*c,
  None),
 (Number Field in c9 with defining polynomial x^8 + 28*x^4 + 2500,
  Ring morphism:
    From: Number Field in c9 with defining polynomial x^8 + 28*x^4 + 2500
    To:   Number Field in c with defining polynomial x^8 + 28*x^4 + 2500
    Defn: c9 |--> c,
  Ring morphism:
    From: Number Field in c with defining polynomial x^8 + 28*x^4 + 2500
    To:   Number Field in c9 with defining polynomial x^8 + 28*x^4 + 2500
    Defn: c |--> c9)
 ]
```

Ten subfields are described, which is what we would expect, given the 10 subgroups of the Galois group. Each begins with a new number field that is a subfield. Technically, each is not a subset of `L`, but the second item returned for each subfield is an injective homomorphism, also known generally as an "embedding." Each embedding describes how a primitive element of the subfield translates to an element of `L`. Some of these primitive elements could be manipulated (as we have done above) to yield slightly simpler minimal polynomials, but the results are quite impressive nonetheless. Each item in the list has a third component, which is almost always `None`, except when the subfield is the whole field, and then the third component is an injective homomorphism "in the other direction."

23.1.4 Normal Extensions

Consider the third subgroup in the list above, generated by the permutation `(1,4)`. As a subgroup of order 2, it only has one nontrivial element, which here corresponds to the seventh automorphism. We determine the fixed elements as before.

```
sage: tau = G[6]
sage: tau_matrix = column_matrix([tau(be).vector() for be in basis])
sage: (tau_matrix-identity_matrix(8)).right_kernel()
Vector space of degree 8 and dimension 4 over Rational Field
Basis matrix:
[    1     0     0     0     0     0     0     0]
[    0     1     0     0     0 -1/82     0     0]
[    0     0     1     0     0     0 -1/22     0]
[    0     0     0     1     0     0     0 11/58]

sage: fromL(tau(1))
1

sage: fromL(tau(c+(-1/82)*c^5))
-120/41*a

sage: fromL(tau(c^2+(-1/22)*c^6))
150/11*a^2

sage: fromL(tau(c^3+(11/58)*c^7))
3000/29*a^3
```

As usual, ignoring rational multiples, we see powers of `a` and recognize that `a` alone will be a primitive element for the fixed field, which is thus $\mathbb{Q}(\sqrt[4]{2})$. Recognize that `a` was our first root of $x^4 - 2$, and was used to create the first part of original tower, `N`. So `N` is both $\mathbb{Q}(\sqrt[4]{2})$ and the fixed field of $H = \langle (1,4) \rangle$.

$\mathbb{Q}(\sqrt[4]{2})$ contains at least one root of the irreducible $x^4 - 2$, but not all of the roots (witness the factorization above) and therefore does not qualify as a normal extension. By part (4) of Theorem 23.15 the automorphism group of the extension is not normal in the full Galois group.

```
sage: sg[2].is_normal(P)
False
```

As expected.

23.2 Exercises

1 In the analysis of Example 7 with Sage, two subgroups of order 2 and one subgroup of order 4 were not analyzed. Determine the fixed fields of these three subgroups.

2 Build the splitting field of $p(x) = x^3 - 6x^2 + 12x - 10$ and then determine the Galois group of $p(x)$ as a concrete group of explicit permutations. Build the lattice of subgroups of the Galois group, again using the same explicit permutations. Using the fundamental theorem of Galois theory, construct the subfields of the splitting field. Include your supporting documentation in your submitted Sage worksheet. Also, submit a written component of this assignment containing a complete layout of the subgroups and subfields, written entirely with mathematical notation and with no Sage commands, designed to illustrate the correspondence between the two. All you need here is the graphical layout, suitably labeled — the Sage worksheet will substantiate your work.

3 The polynomial $x^5 - x - 1$ has all of the symmetric group S_5 as its Galois group. Because S_5 is not solvable, we know this polynomial to be an example of a quintic polynomial that is not solvable by radicals. Unfortunately, asking Sage to compute this Galois group takes far too long. So this exercise will simulate that experience with a slightly smaller example.

Consider the polynomial $p(x) = x^4 + x + 1$.
(a) Build the splitting field of $p(x)$ one root at a time. Create an extension, factor there, discard linear factors, use the remaining irreducible factor to extend once more. Repeat until $p(x)$ factors completely. Be sure to do a final extension via just a linear factor. This is a little silly, and Sage will seem to ignore your final generator (so you will want to setermine what it is equivalent to in terms of the previous gfenerators). Directions below depend on taking this extra step.
(b) Factor the original polynomial over the final extension field in the tower. What is boring about this factorization in comparison to some other examples we have done?
(c) Construct the full tower as an absolute field over $\mathbb{Q}$. From the degree of this extension and the degree of the original polynomial, infer the Galois group of the polynomial.
(d) Using the mappings that allow you to translate between the tower and the absolute field (obtained from the `.structure()` method), choose one of the roots (any one)

and express it in terms of the single generator of the absolute field. Then reverse the procedure and express the single generator of the absolute field in terms of the roots in the tower.

(e) Compute the group of automorphisms of the absolute field (but don't display the whole group in what you submit). Take all four roots (including your silly one from the last step of the tower construction) and apply each field automorphism to the four roots (creating the guaranteed permutations of the roots). Comment on what you see.

(f) There is one nontrivial automorphism that has an especially simple form (it is the second one for me) when applied to the generator of the absolute field. What does this automorphism do to the roots of $p(x)$?

(g) Consider the extension of $\mathbb{Q}$ formed by adjoining just one of the roots. This is a subfield of the splitting field of the polynomial, so is the fixed field of a subgroup of the Galois group. Give a simple description of the corresponding subgroup using language we typically only apply to permutation groups.

4 Return to the splitting field of the quintic discussed in the introduction to the previous problem ($x^5 - x - 1$). Create the first two intermediate fields by adjoining two roots (one at a time). But instead of factoring at each step to get a new irreducible polynomial, *divide* by the linear factor you *know* is a factor. In general, the quotient might factor further, but in this exercise presume it does not. In other words, act as if your quotient by the linear factor is irreducible. If it is not, then the `NumberField` command should complain (which it won't).

After adjoining two roots, create the extension producing a third root, and do the division. You should now have a quadratic factor. Assuming the quadratic is irreducible (it is) argue that you have enough evidence to establish the order of the Galois group, and hence can determine *exactly* which group it is.

You can try to use this quadratic factor to create one more step in the extensions, and you will arrive at the splitting field, as can be seen with logic or division. However, this could take a long time to complete (save your work beforehand!). You can try passing the `check=False` argument to the `NumberField` command — this will bypass checking irreducibility.

5 Create the finite field of order 3^6, letting Sage supply the default polynomial for its construction. The polynomial $x^6 + x^2 + 2 * x + 1$ is irreducible over this finite field. Check that this polynomial splits in the finite field, and then use the `.roots()` method to collect the roots of the polynomial. Get the group of automorphisms of the field with the `End()` command.

You now have all of the pieces to associate each field automorphism with a permutation of the roots. From this, identify the Galois group and all of its subgroups. For each subgroup, determine the fixed field. You might find the roots easier to work with if you use the `.log()` method to identify them as powers of the field's multiplicative generator.

Your Galois group in this example will be abelian. So every subgroup is normal, and hence any extension is also normal. Can you give extend this example by choosing a nontrivial intermediate field with a nontrivial irreducible polynomial that has all of its roots in the intermediate field and a nontrivial irreducible polynomial with none of its roots in the intermediate field? Your results here are "typical" in the sense that the particular field or irreducible polynomial makes little difference in the qualitative nature of the results.

6 The splitting field for the irreducible polynomial $p(x) = x^7 - 7x + 3$ has degree 168 (hence this is the order of the Galois group). This polynomial is derived from an "Elkies trinomial curve," a hyperelliptic curve (below) that produces polynomials with interesting Galois groups:

$$y^2 = x(81x^5 + 396x^4 + 738x^3 + 660x^2 + 269x + 48)$$

For $p(x)$ the resulting Galois group is $PSL(2, 7)$, a simple group. If $SL(2, 7)$ is all 2×2 matrices over $\mathbb{Z}_7$ with determinant 1, then $PSL(2, 7)$ is the quotient by the subgroup $\{I_2, -I_2\}$. It is the second-smallest non-abelian simple group (after A_5).

See how far you can get in using Sage to build this splitting field. A degree 7 extension will yield one linear factor, and a subsequent degree 6 extension will yield two linear factors, leaving a quartic factor. Here is where the computations begin to slow down. If we believe that the splitting field has degree 168, then we know that adding a root from this degree 4 factor will get us to the splitting field. Creating this extension may be possible computationally, but verifying that the quartic splits into linear factors here seems to be infeasible.

7 Return to 7, and the complete list of subfields obtainable from the `.subfields()` method applied to the flattened tower. As mentioned, these are technically not subfields, but do have embeddings into the tower. Given two subfields, their respective primitive elements are embedded into the tower, with an image that is a linear combination of powers of the primitive element for the tower.

If one subfield is contained in the other, then the image of the primitive element for the smaller field should be a linear combination of the (appropriate) powers of the image of the primitive element for the larger field. This is a linear algebra computation that should be possible in the tower, relative to the power basis for the whole tower.

Write a procedure to determine if two subfields are related by one being a subset of the other. Then use this procedure to create the lattice of subfields. The eventual goal would be a graphical display of the lattice, using the existing plotting facilities available for lattices, similar to the top half of Figure 23.3. This is a "challenging" exercise, which is code for "it has not been tested."

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. Applicability And Definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other

implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. Copying In Quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely

this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. Collections Of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. Future Revisions Of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Addendum: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.