Terminal commands before running:

pip install prettytable #used to create cayley tables

pip install itertools #used to generate subsets

pip install graphviz #used to create cayley graphs

brew install graphviz #installs graphviz to system

Algebra

This project is used to help the user understand certain finite groups in abstract algebra better to aid in the process of learning, or just for quick reference when necessary.

The program logic for creating groups and working with them is all based on the actual definitions and theorems from group theory that can be proven.

I began with the concept of a binary structure, which is a set with a given closed binary operation. In the binary structure class are mostly methods that test whether a given set and operation satisfies the axioms of a group.

The methods of the BinStruct class check for associativity, check for an identity element, and check for inverses. These are three fundamental conditions for a binary structure to be a group. Associativity checks every possible ordered triple (a, b, c) to see if (a \* b) \* c = a \* (b \* c). The identity method looks in the set for an element e with the property that for any x, e \* x = x and x \* e = x. There is also a method to return the identity, if one exists. For inverses, the method checks each element x, looking for an inverse y such that x \* y = e and y \* x = e. Again, there is a method to return the inverse for any given element.

I was also able to include the Cayley table functionality in the binary structure class because there is nothing special about the group axioms that allow only groups to be described by Cayley tables and not general binary structures.

The Cayley table method uses the functionality of the prettytable package. This package allows the easy creation of tables row by row, so all I had to do was iteratively create lists with list comprehensions that calculate each value.

The Group class, which is a subclass of BinStruct, has an \_\_init\_\_ method that makes sure that the set and rule given do actually constitute a group before anything else is done. Otherwise, an exception is raised. The constructor also has the functionality of creating a map between the elements of the set and their orders in the group. Finally, it creates a list of all the generators of the Group, elements with orders equal to the order of the group.

The method used to find the order of each element does it the obvious way: by taking the element to increasing powers until the identity is reached.

A major method in the Group class is the one that finds all the proper subgroups of a given group. It does so by, for each divisor of the order of the group, finding all the subsets of that order that contain the identity (this is an optimization based on the fact that the identity is in every subgroup). These subsets are found using the combination functionality of the itertools package. Once the subsets are found, we check if the subset is a subgroup under the same operation by using the one-step subgroup test (check if for each pair of elements of the subset, the first “times” the inverse of the second remains in the subset).

There are also methods to check if a group is abelian or cyclic. The abelian test literally just checks if every pair of elements commute. The cyclic test checks if the group has any generators.

I chose to implement the \_\_eq\_\_ method as an isomorphism test. The way the test works is very roundabout: it creates the symmetric group of the size of the two groups, and checks for all possible permutations of isomorphism based on position in the list and which symmetry is chosen. It essentially only needs to check for homomorphisms because if the two sets are not the same size, the method immediately returns False.

The final method of the Group class is the cayley\_graph method, which creates a png image of a Cayley graph for the group. I used the graphviz library to create the graph. The method adds each element of the set to the graph as a node, and then for each element passed as a parameter to a method (referred to as “generators” even though they do not individually generate the group; the idea being they should generate the group together) checks for each pair of elements in the group if the first “times” the generator equals the second. If it does, a directed edge is drawn, labeled with which generator has been chosen.

There are a few functions in the group.py file that are not part of the Group class. These are the functions that actually create the interesting finite groups.

There are functions used to create additive and multiplicative integer groups. For additive groups, we just define the set as the range of non-negative integers up to (not including) the value given and the rule as addition mod the value. For multiplicative groups, the set created using a comprehension that takes all the positive integers less than and relatively prime to the value and the rule is multiplication mod the value.

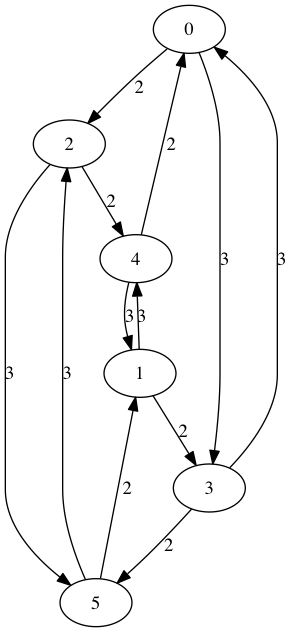
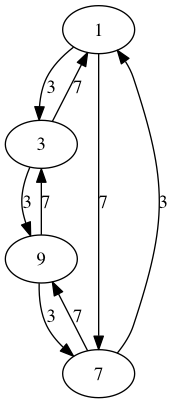
Next, I needed a way to create permutation groups, so I created a Permutation class that is initiated from an array of what the permutation looks like. Its methods are \_\_str\_\_ to represent it in cycle notation, \_\_mul\_\_ to compose two permutations, \_\_eq\_\_ for comparison, and a custom \_\_hash\_\_ function to make sure equality follows the rules. Finally, I created an is\_even method to help in the construction of alternating groups.

Back in group.py, there is a recursive method to generate all n! permutations of n objects, which allows for easy creation of the symmetric and alternating groups. The symmetric group’s set is created with a list comprehension of the permutations generated by the code discussed above. As stated above, for the alternating groups, we create them the same way, just check if each element is even. The rules for both of these were easy to create because of the custom multiplication I created.

As I was working on this project, I was surprised by how many issues arise when trying to translate what seems to be very formal, specific mathematic language into computer science. Mathematicians take a lot about language for granted that is hard to express simply in computer science. For example, I did not expect to have to use the symmetric group in order to define isomorphism.

I plan to expand this project further. The isomorphism and order functionality go largely unused in the present runnable state, and there are definitely things to add that could use these. The ability to use \_\_eq\_\_ to consider things equal “up to isomorphism” seems like a really interesting concept to continue working with.

The plan is to add functionality for rings and fields, and maybe even modules and algebras, in the future. What would be really cool is the ability to somehow represent infinite groups (and thus infinite rings, fields, etc.) but I would likely have to figure out how to map that general concept onto the idea of floating point numbers, which seems rather complicated. Finite algebra, while still complex, seems more rigorously representable, at least until we get quantum computers.

Have some nice Cayley graphs:

