GAP NOTES

WILLIAM J. DEMEO

Contents

Date: 2 June 2011.

1. Introduction: a very few, very basic commands

a. **Help.** A convenient way of reading the documentation at the gap prompt is with the `?` symbol, as in:

```
?tutorial:a first session with gap
```

This starts the basic GAP tutorial. After reading the first page, you can move to the next page by entering `?>`, and the previous page with `?<`.

As another example, if you want to learn about commands involving, say, the centralizer, just enter the command `?centralizer` for a list of related help topics.

b. **Comments** in GAP begin with the sharp character `#`. The whole comment, including `#` and the newline character, is treated as a single whitespace.

c. **Functions.** You can define a function in a `.gap` file or "in-line" as follows:

```
cubed := x -> x^3;
```

Then `cubed(5)` returns 125.

d. GAP composes permutations left-to-right. That is, `(1,2,4)*(1,2)` gives `(2,4)`; *not* right-to-left; that is, $(1,2,4)(1,2) \neq (1,4)$.

e. The carrot operator `^` is used to determine the image of a point under a permutation and to conjugate one permutation by another. e.g. `(1,2,3)^-1` gives `(1,3,2)`, `2^(1,2,3)` gives 3, and `(1,2,3)^(1,2)` gives `(1,3,2)`. That is (for the last one), $(1,2,3)^{(1,2)} = (1,2)(1,2,3)(1,2) = (1,3,2)$.

f. The preimage of a point i under a permutation p is given by i/p. For example, `2/(1,2,3)` gives 1, and, since $(1,2,4)(1,2) = (2,4)$, `3/((1,2,4)(1,2))` gives 3, and `2/((1,2,4)(1,2))` gives 4.

g. You can get an overview of all GAP variables with the command `NamesGVars();` and you can see the user-defined variables of the current session using `NamesUserGVars();`. The name of every global variable in the GAP library starts with a capital letter, so you should start user-defined variable names with a lower case letter (or a number).

2. Some important groups[8]

There are several infinite families of groups which are parametrized by numbers. GAP provides various functions to construct these groups. The functions always permit (but do not require) one to indicate a filter, for example `IsPermGroup, IsMatrixGroup` or `IsPcGroup`, in which the group shall be constructed. There always is a default filter corresponding to a "natural" way to describe the group in question. Note that not every group can be constructed in every filter, there may be theoretical restrictions (`IsPcGroup` only works for solvable groups) or methods may be available only for a few filters. Certain filters may admit additional hints. For example, groups constructed in `IsMatrixGroup` may be constructed over a specified field, which can be given as second argument of the function that constructs the group; The default field is `Rationals`.

- `TrivialGroup( [filt] )`

  constructs a trivial group in the category given by the filter `filt`. If `filt` is not given it defaults to `IsPcGroup`.

  ```
  gap> TrivialGroup();
  <pc group of size 1 with 0 generators>
  gap> TrivialGroup( IsPermGroup );    # returns Group(())
  ```

  CAUTION: if you want to test whether a certain subgroup $H \leqslant G$ is the identity of G, you can't simply type `H=Group(())` or `H=TrivialGroup()`. Instead, define the trivial subgroup of G with `e:=Group([Identity(G)])`. Then `H=e` tests whether H is trivial.

- `CyclicGroup( [filt, ] n )`

  constructs the cyclic group of size n in the category `filt`. If `filt` is not given it defaults to `IsPcGroup`.

  ```
  gap> CyclicGroup(12);
  <pc group of size 12 with 3 generators>
  gap> CyclicGroup(IsPermGroup,12);
  Group([ (1,2,3,4,5,6,7,8,9,10,11,12) ])
  gap> matgrp1:= CyclicGroup( IsMatrixGroup, 12 );
  <matrix group of size 12 with 1 generators>
  gap> FieldOfMatrixGroup( matgrp1 );
  Rationals
  gap> matgrp2:= CyclicGroup( IsMatrixGroup, GF(2), 12 );
  <matrix group of size 12 with 1 generators>
  gap> FieldOfMatrixGroup( matgrp2 );
  GF(2)
  ```

- `AbelianGroup( [filt, ] i )`

  constructs an abelian group in the category `filt` which is of isomorphism type

  $$\mathbb{Z}_{\mathtt{i}[1]} \oplus \mathbb{Z}_{\mathtt{i}[2]} \oplus \cdots \oplus \mathbb{Z}_{\mathtt{i}[n]}.$$

  Here `i` must be a list of positive integers. If `filt` is not given it defaults to `IsPcGroup`. The generators of the group returned corresponding to the integers in `i`.

  ```
  gap> AbelianGroup([1,2,3]);
  ```

[8] See also the GAP Manual [3], Chapter 48 "Group Libraries."

```
<pc group of size 6 with 3 generators>
```

- `ElementaryAbelianGroup( [filt, ] n )`

  constructs the elementary abelian group of size n in the category given by the filter `filt`. If `filt` is not given it defaults to `IsPcGroup`.

```
gap> ElementaryAbelianGroup(8192);
<pc group of size 8192 with 13 generators>
```

- `DihedralGroup( [filt, ] n )`

  constructs the dihedral group of size n in the category given by the filter `filt`. If `filt` is not given it defaults to `IsPcGroup`.

```
gap> DihedralGroup(10);
<pc group of size 10 with 2 generators>
```

- `AlternatingGroup( [filt, ] deg )`
- `AlternatingGroup( [filt, ] dom )`

  constructs the alternating group of degree `deg` in the category given by the filter `filt`. If `filt` is not given it defaults to `IsPermGroup`. In the second version, the function constructs the alternating group on the points given in the set `dom` which must be a set of positive integers.

```
gap> AlternatingGroup(5);
Alt( [ 1 .. 5 ] )
```

- `SymmetricGroup( [filt, ] deg )`
- `SymmetricGroup( [filt, ] dom )`

  constructs the symmetric group of degree `deg` in the category given by the filter `filt`. If `filt` is not given it defaults to `IsPermGroup`. In the second version, the function constructs the symmetric group on the points given in the set `dom` which must be a set of positive integers.

```
gap> SymmetricGroup(10);
Sym( [ 1 .. 10 ] )
```

3. Factor groups[9]

- `NaturalHomomorphismByNormalSubgroup( G, N )`
- `NaturalHomomorphismByNormalSubgroupNC( G, N )`

  returns a homomorphism from G to another group whose kernel is N. GAP will try to select the image group as to make computations in it as efficient as possible. As the factor group G/N can be identified with the image of G this permits efficient computations in the factor group. The homomorphism returned is not necessarily surjective, so `ImagesSource` should be used instead of `Range` to get a group isomorphic to the factor group. The `NC` variant does not check whether N is normal in G.
- `FactorGroup( G, N )`, `FactorGroupNC( G, N )`

  returns the image of the `NaturalHomomorphismByNormalSubgroup(G,N)`. The `NC` version does not test whether N is normal in G.

```
gap> g:=Group((1,2,3,4),(1,2));;  n:=Subgroup(g,[(1,2)(3,4),(1,3)(2,4)]);;
gap> hom:=NaturalHomomorphismByNormalSubgroup(g,n);
[ (1,2,3,4), (1,2) ] -> [ f1*f2, f1 ]
gap> Size(ImagesSource(hom));  # returns 6
gap> FactorGroup(g,n);         # returns Group([ f1, f2 ])
```

- `CommutatorFactorGroup( G )`

  computes the commutator factor group G/G' of the group G.

```
gap> CommutatorFactorGroup(g);
Group([ f1 ])
```

- `MaximalAbelianQuotient( grp )`

  returns an epimorphism from `grp` onto the maximal abelian quotient of `grp`. The kernel of this epimorphism is the derived subgroup.
- `HasAbelianFactorGroup( G, N )`

  tests whether G/N is abelian (without explicitly constructing the factor group).
- `HasElementaryAbelianFactorGroup( G, N )`

  tests whether G/N is elementary abelian (without explicitly constructing the factor group).

```
gap> HasAbelianFactorGroup(g,n);                    # returns  false
gap> HasAbelianFactorGroup(DerivedSubgroup(g),n);   # returns  true
```

- `CentralizerModulo( G, N, x )`

  computes the full preimage of the centralizer $C_{G/N}(\mathrm{x} \cdot N)$ in G (without necessarily constructing the factor group).

```
gap> CentralizerModulo(g,n,(1,2));
Group([ (3,4), (1,3)(2,4), (1,4)(2,3) ])
```

[9] See also the GAP Manual [3], Section 37.18.

4. Some important subgroups

In the following, we refer to "magmas" which are more general than groups. If you are not familiar with magmas, just substitute the word group for the word magma.

- `Normalizer( G, U )`
- `Normalizer( G, g )`

  Computes the normalizer $N_G(U)$, that is the stabilizer of U under the conjugation action of G. The second form computes $N_G(\langle g \rangle)$.

```
gap> Normalizer(g,Subgroup(g,[(1,2,3)]));  # returns Group([ (1,2,3), (2,3) ])
```

- `Centralizer( M, s )`
- `Centralizer( M, S )`

  For an element s of the magma M this operation returns the centralizer of s. This is the domain of those elements m in M that commute with s. For a submagma S it returns the domain of those elements that commute with all elements s of S.

- `Center( M )` Center returns the center of the magma M; i.e., the domain of those elements m in M that commute and associate with all elements of M. For associative magmas we have that Center(M) = Centralizer(M, M). The center of a magma is always commutative.
- `CommutatorSubgroup( G, H )`

  If G and H are two groups of elements in the same family, this operation returns the group generated by all commutators $[g,h] = ghg^{-1}h^{-1}$ of elements $g \in G$ and $h \in H$; that is, the group $\langle [g,h] \mid g \in G, h \in H \rangle$.

```
gap> CommutatorSubgroup(Group((1,2,3),(1,2)),Group((2,3,4),(3,4)));
Group([ (1,4)(2,3), (1,3,4) ])
gap> Size(last);              # returns 12
```

- `DerivedSubgroup( G )`

  The derived subgroup G' of G is the subgroup generated by all commutators of pairs of elements of G. It is normal in G and the factor group G/G' is the largest abelian factor group of G. If G/H is abelian, then $G' \leqslant H$.

- `Core( S, U )`

  If S and U are groups of elements in the same family, this operation returns the core of U in S, that is the intersection of all S-conjugates of U.

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> Core(g,Subgroup(g,[(1,2,3,4)]));    # returns Group(())
```

- `PCore( G, p )`

  The p-core of G is the largest normal p-subgroup of G. It is the core of a p-Sylow subgroup of G.

4.1. **Sets of Subgroups.**

- `ConjugacyClassSubgroups( G, U )`
  generates the conjugacy class of subgroups of G with representative U. This class is an external set, so functions such as `Representative`, (which returns U), `ActingDomain` (which returns G), `StabilizerOfExternalSet` (which returns the normalizer of U), and `AsList` work for it.

  (The use of `[]` list access to select elements of the class is considered obsolete and will be removed in future versions. Use `ClassElementLattice` instead.)

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> IsNaturalSymmetricGroup(g);   # returns true
gap> cl:=ConjugacyClassSubgroups( g, Subgroup(g,[(1,2)]) );
Group( [ (1,2) ] )^G
gap> Size(cl);                     # returns 6
gap> ClassElementLattice( cl, 4 ); # returns Group([ (2,3) ])
```

- `IsConjugacyClassSubgroupsRep( obj )`
- `IsConjugacyClassSubgroupsByStabilizerRep( obj )`

  Is the representation GAP uses for conjugacy classes of subgroups. It can be used to check whether an object is a class of subgroups. The second function `...ByStabilizerRep` is, in addition, an external orbit by stabilizer and will compute its elements via a transversal of the stabilizer.
- `ConjugacyClassesSubgroups( G )`
  This attribute returns a list of all conjugacy classes of subgroups of the group G. It also is applicable for lattices of subgroups. The order in which the classes are listed depends on the method chosen by GAP. For each class of subgroups, a representative can be accessed using `Representative`.

```
gap> ConjugacyClassesSubgroups( g );
[ Group( () )^G, Group( [ (1,3)(2,4) ] )^G, Group( [ (3,4) ] )^G,
Group( [ (2,4,3) ] )^G, Group( [ (1,4)(2,3), (1,3)(2,4) ] )^G,
Group( [ (1,2)(3,4), (3,4) ] )^G, Group( [ (1,2)(3,4), (1,3,2,4) ] )^G,
Group( [ (3,4), (2,4,3) ] )^G, Group( [ (1,3)(2,4), (1,4)(2,3), (1,2) ] )^G,
Group( [ (1,3)(2,4), (1,4)(2,3), (2,4,3) ] )^G,
Group( [ (1,3)(2,4), (1,4)(2,3), (2,4,3), (1,2) ] )^G ]
```

- `ConjugacyClassesMaximalSubgroups( G )`
  returns the conjugacy classes of maximal subgroups of G. Representatives of the classes can be computed directly by `MaximalSubgroupClassReps` (see next item).

```
gap> ConjugacyClassesMaximalSubgroups( g );
[ AlternatingGroup( [ 1 .. 4 ] )^G, Group( [ (1,2,3), (1,2) ] )^G,
Group( [ (1,2), (3,4), (1,3)(2,4) ] )^G ]
```

- `MaximalSubgroupClassReps( G )` returns a list of conjugacy representatives of the maximal subgroups of G.

```
gap> MaximalSubgroupClassReps( g );
[ Alt( [ 1 .. 4 ] ), Group([ (1,2,3), (1,2) ]),
Group([ (1,2), (3,4), (1,3)(2,4) ]) ]
```

- `MaximalSubgroups( G )`
  returns a list of all maximal subgroups of G. This may take up much space, therefore the

command should be avoided if possible (e.g., by using `ConjugacyClassesMaximalSubgroups( G )` instead.)

```
gap> MaximalSubgroups( Group((1,2,3),(1,2)) );
[ Group([ (1,2,3) ]), Group([ (2,3) ]), Group([ (1,2) ]), Group([ (1,3) ]) ]
```

- `NormalSubgroups( G )`
  returns a list of all normal subgroups of G.

```
gap> g := SymmetricGroup( 4 );;  NormalSubgroups( g );
[ Group(()), Group([ (1,4)(2,3), (1,3)(2,4) ]),
Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]), Sym( [ 1 .. 4 ] ) ]
```

- `MaximalNormalSubgroups( G )`
  is a list containing those proper normal subgroups of the group G that are maximal among the proper normal subgroups.

```
gap> MaximalNormalSubgroups( g );
[ Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]) ]
```

- `MinimalNormalSubgroups( G )`
  is a list containing those nontrivial normal subgroups of the group G that are minimal among the nontrivial normal subgroups.

```
gap> MinimalNormalSubgroups( g );  # returns [ Group([ (1,4)(2,3), (1,3)(2,4) ]) ]
```

4.2. **Subgroup Lattice.** There is a very useful GAP package called XGap which produces nice graphical displays of subgroup lattices. If you have XGap installed and running, at the command prompt you can try, for example:

```
gap> g := SymmetricGroup( 4 );;
gap> GraphicSubgroupLattice( g );
```

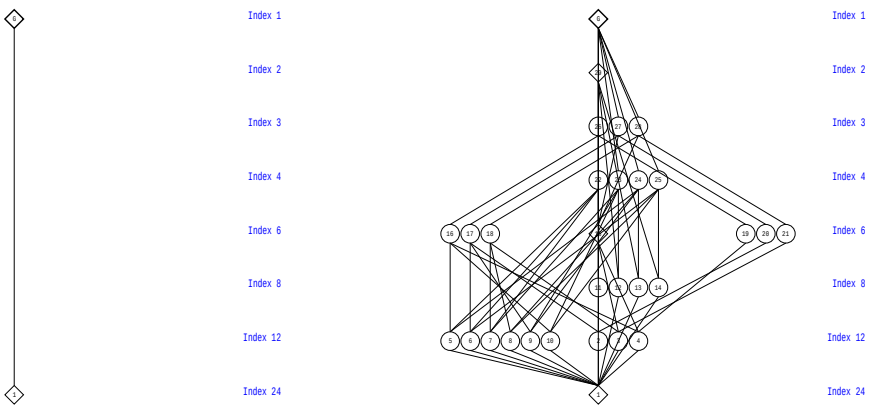


FIGURE 1. The subgroups (e) and S_4 (left), and the full subgroup lattice Sub[$S4$] (right).

This opens a new XGap graphics window showing just the two subgroups (e) and S_4 – Figure 1 (left). In the `Subgroups` menu of this new window, select `All Subgroups`. This fills in the subgroup lattice Sub[S_4] (in a rather messy way) – Figure 1 (right). You can then move the subgroups around to make it look nicer – Figure 2.

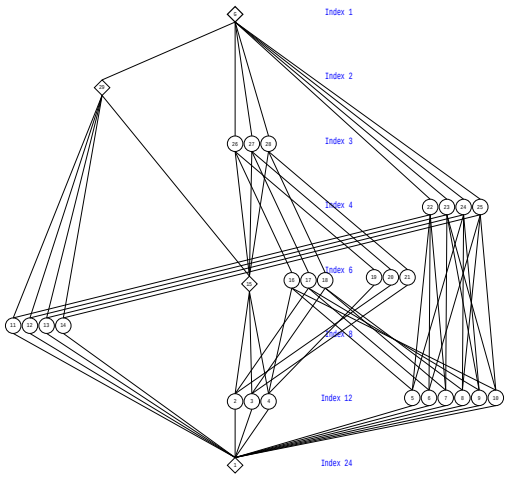


FIGURE 2. Sub[$S4$] as drawn by the XGap program.

A couple of nice features of the XGap display of the subgroup lattice is that conjugacy classes of subgroups are glued together, and their indices are displayed. Also, normal subgroups are drawn as diamonds, while non-normal subgroups appear as circles. Occasionally, when the group is very large, XGap does not waste time checking which subgroups are normal, and instead displays them with squares.

We now list some subgroup lattice commands that are available in the standard GAP distribution. (These do not require XGap.)

- `LatticeSubgroups( G )`
  computes the lattice of subgroups of the group G. This lattice has the conjugacy classes of subgroups as attribute `ConjugacyClassesSubgroups` and permits one to test maximality/minimality relations.

```
gap> g := SymmetricGroup( 4 );;  l := LatticeSubgroups( g );
<subgroup lattice of Sym( [ 1 .. 4 ] ), 11 classes, 30 subgroups>
gap> ConjugacyClassesSubgroups( l );
[ Group( () )^G, Group( [ (1,3)(2,4) ] )^G, Group( [ (3,4) ] )^G,
Group( [ (2,4,3) ] )^G, Group( [ (1,4)(2,3), (1,3)(2,4) ] )^G,
Group( [ (1,2)(3,4), (3,4) ] )^G, Group( [ (1,2)(3,4), (1,3,2,4) ] )^G,
Group( [ (3,4), (2,4,3) ] )^G, Group( [ (1,3)(2,4), (1,4)(2,3), (1,2) ] )^G,
Group( [ (1,3)(2,4), (1,4)(2,3), (2,4,3) ] )^G,
Group( [ (1,3)(2,4), (1,4)(2,3), (2,4,3), (1,2) ] )^G ]
```

- `ClassElementLattice( C, n )`
  For a class C of subgroups, obtained by a lattice computation, this operation returns the n-th conjugate subgroup in the class. *Because of other methods installed,* `AsList(C)` *can give a different arrangement of the class elements!*

- `MaximalSubgroupsLattice( lat )`
  For a lattice `lat` of subgroups this attribute contains the maximal subgroup relations among the subgroups of the lattice. It is a list, corresponding to the ConjugacyClassesSubgroups of the lattice, each entry giving a list of the maximal subgroups of the representative of this class. Every maximal subgroup is indicated by a list of the form `[cls, nr]` which means that the `nr`-th subgroup in class number `cls` is a maximal subgroup of the representative. *The number* `nr` *corresponds to access via* `ClassElementLattice` *and not necessarily the* `AsList` *arrangement!*

```
gap> MaximalSubgroupsLattice(l);
[ [ ], [ [ 1, 1 ] ], [ [ 1, 1 ] ], [ [ 1, 1 ] ],
[ [ 2, 1 ], [ 2, 2 ], [ 2, 3 ] ], [ [ 3, 1 ], [ 3, 6 ], [ 2, 3 ] ],
[ [ 2, 3 ] ], [ [ 4, 1 ], [ 3, 1 ], [ 3, 2 ], [ 3, 3 ] ],
[ [ 7, 1 ], [ 6, 1 ], [ 5, 1 ] ],
[ [ 5, 1 ], [ 4, 1 ], [ 4, 2 ], [ 4, 3 ], [ 4, 4 ] ],
[ [ 10, 1 ], [ 9, 1 ], [ 9, 2 ], [ 9, 3 ], [ 8, 1 ], [ 8, 2 ], [ 8, 3 ],
[ 8, 4 ] ] ]
gap> last[6];   # returns [ [ 3, 1 ], [ 3, 6 ], [ 2, 3 ] ]
gap> u1:=Representative(ConjugacyClassesSubgroups(l)[6]);
Group([ (1,2)(3,4), (3,4) ])
gap> u2:=ClassElementLattice(ConjugacyClassesSubgroups(l)[3],1);;
gap> u3:=ClassElementLattice(ConjugacyClassesSubgroups(l)[3],6);;
gap> u4:=ClassElementLattice(ConjugacyClassesSubgroups(l)[2],3);;
gap> IsSubgroup(u1,u2); IsSubgroup(u1,u3); IsSubgroup(u1,u4);
# returns true, true, true
```

- `MinimalSupergroupsLattice( lat )`
  For a lattice `lat` of subgroups this attribute contains the minimal supergroup relations among the subgroups of the lattice. It is a list, corresponding to the ConjugacyClassesSubgroups of the lattice, each entry giving a list of the minimal supergroups of the representative of this class. As above, every minimal supergroup is indicated by a list of form `[cls, nr]` – the `nr`-th subgroup in class number `cls` is a minimal supergroup of the representative.

```
gap> MinimalSupergroupsLattice(l);
[ [ [ 2, 1 ], [ 2, 2 ], [ 2, 3 ], [ 3, 1 ], [ 3, 2 ], [ 3, 3 ], [ 3, 4 ],
[ 3, 5 ], [ 3, 6 ], [ 4, 1 ], [ 4, 2 ], [ 4, 3 ], [ 4, 4 ] ],
[ [ 5, 1 ], [ 6, 2 ], [ 7, 2 ] ], [ [ 6, 1 ], [ 8, 1 ], [ 8, 3 ] ],
[ [ 8, 1 ], [ 10, 1 ] ], [ [ 9, 1 ], [ 9, 2 ], [ 9, 3 ], [ 10, 1 ] ],
[ [ 9, 1 ] ], [ [ 9, 1 ] ], [ [ 11, 1 ] ], [ [ 11, 1 ] ], [ [ 11, 1 ] ],
[ ] ]
gap> last[3];   # returns  [ [ 6, 1 ], [ 8, 1 ], [ 8, 3 ] ]
gap> u5:=ClassElementLattice(ConjugacyClassesSubgroups(l)[8],1);
Group([ (3,4), (2,4,3) ])
gap> u6:=ClassElementLattice(ConjugacyClassesSubgroups(l)[8],3);
Group([ (1,3), (1,3,4) ])
```

5. Subgroup series[10]

In group theory many subgroup series are considered, and GAP provides commands to compute them. In the following sections, there is always a series

$$G = U_1 > U_2 > \cdots > U_m = \langle 1 \rangle$$

of subgroups considered. A series also may stop without reaching G or $\langle 1 \rangle$.

A series is called *subnormal* if every U_{i+1} is normal in U_i.

A series is called *normal* if every U_i is normal in G.

A series of normal subgroups is called *central* if U_i/U_{i+1} is central in G/U_{i+1}.

We call a series *refinable* if intermediate subgroups can be added to the series without destroying the properties of the series. Unless explicitly declared otherwise, all subgroup series are descending. That is, they are stored in decreasing order.

- `ChiefSeries( G )`

  is a series of normal subgroups of G which cannot be refined further. That is, there is no normal subgroup $N \trianglelefteq G$ with $U_i > N > U_{i+1}$. This attribute returns one chief series (of potentially many possibilities).

  ```
  gap> g:=Group((1,2,3,4),(1,2));;
  gap> ChiefSeries(g);
  [ Group([ (1,2,3,4), (1,2) ]), Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
  Group([ (1,4)(2,3), (1,3)(2,4) ]), Group(()) ]
  ```

- `ChiefSeriesThrough( G, L )`

  is a chief series of the group G going through the normal subgroups in the list L. Here L must be a list of normal subgroups of G contained in each other, sorted by descending size. This attribute returns one chief series (of potentially many possibilities).

- `ChiefSeriesUnderAction( H, G )`

  returns a series of normal subgroups of G which are invariant under H such that the series cannot be refined any further. G must be a subgroup of H. This attribute returns one such series (of potentially many possibilities).

- `SubnormalSeries( G, U )`

  If U is a subgroup of G this operation returns a subnormal series that descends from G to a subnormal subgroup V containing U. If U is subnormal, V=U.

  ```
  gap> s:=SubnormalSeries(g,Group((1,2)(3,4)));
  [ Group([ (1,2,3,4), (1,2) ]), Group([ (1,2)(3,4), (1,4)(2,3) ]),
  Group([ (1,2)(3,4) ]) ]
  ```

- `CompositionSeries( G )`

  A *composition series* is a subnormal series which cannot be refined. This attribute returns one composition series (of potentially many possibilities).

- `DisplayCompositionSeries( G )`

  Displays a composition series of G in a nice way, identifying the simple factors.

[10] See also the GAP Manual [3], page 370.

```
gap> CompositionSeries(g);
[ Group([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
Group([ (1,4)(2,3), (1,3)(2,4) ]), Group([ (1,3)(2,4) ]), Group(()) ]
gap> DisplayCompositionSeries(Group((1,2,3,4,5,6,7),(1,2)));
G (2 gens, size 5040)
| Z(2)
S (5 gens, size 2520)
| A(7)
1 (0 gens, size 1)
```

- `DerivedSeriesOfGroup( G )`

  The derived series of a group is obtained by $U_{i+1} = U_i'$. It stops if U_i is perfect.[11]
- `DerivedLength( G )`

  The *derived length* of a group is the number of steps in the derived series. (As there is always the group, it is the series length minus 1.)

```
gap> List(DerivedSeriesOfGroup(g),Size);
[ 24, 12, 4, 1 ]
gap> DerivedLength(g);      # returns 3
```

- `AscendingChain( G, U )`

  This function computes an ascending chain of subgroups from U to G. This chain is given as a list whose first entry is U and the last entry is G. The function tries to make the links in this chain small. The option `refineIndex` can be used to give a bound for refinements of steps to avoid GAP trying to enforce too small steps.
- `IntermediateGroup( G, U )`

  This routine tries to find a subgroup E of G, such that $G > E > U$. If U is maximal, it returns `fail`. This is done by finding minimal blocks for the operation of G on the right cosets of U.
- `IntermediateSubgroups( G, U )`

  returns a list of all subgroups of G that properly contain U; that is all subgroups between G and U. It returns a record with components `subgroups` which is a list of these subgroups as well as a component `inclusions` which lists all maximality inclusions among these subgroups. A maximality inclusion is given as a list `[i,j]` indicating that subgroup number `i` is a maximal subgroup of subgroup number `j`, the numbers 0 and 1+length(`subgroups`) are used to denote U and G respectively. (See Section 10 below for a concrete example.)

[11] Recall, a group G is called *perfect* if $G' = G$. Equivalently, G has no nontrivial abelian factor group G/H (since G/G' is the largest abelian factor group, and, if G/H is abelian, then $G' \leqslant H$).

6. Mappings and relations in GAP[12]

As usual, a *relation* is a set of ordered pairs; i.e. a subset of a Cartesian product. For a relation $\mathscr{R}$, we define the *domain* of $\mathscr{R}$ (denoted $\operatorname{dom}\mathscr{R}$) and the *range* of $\mathscr{R}$ ($\operatorname{ran}\mathscr{R}$) by

$$\begin{aligned} x \in \operatorname{dom}\mathscr{R} \quad &\Leftrightarrow \quad \exists y\ (x,y) \in \mathscr{R}, \\ x \in \operatorname{ran}\mathscr{R} \quad &\Leftrightarrow \quad \exists t\ (t,x) \in \mathscr{R}, \end{aligned}$$

In other words,

$$\operatorname{dom}\mathscr{R} = \{x \mid \exists\, y\ (x,y) \in \mathscr{R}\} \quad \text{and} \quad \operatorname{ran}\mathscr{R} = \{y \mid \exists\, x\ (x,y) \in \mathscr{R}\}.$$

A *function* is a relation $\mathscr{F}$ such that for each x in $\operatorname{dom}\mathscr{F}$ there is only one y such that $(x,y) \in \mathscr{F}$.

In GAP, relations are called *generalized mapping*, and functions are called *mappings*. Most GAP operations are declared for general mappings. A general mapping m in GAP is described by its source S, its range R, and a subset $\mathscr{R}$ of the direct product $S \times R$, which is called the underlying relation of m. The objects S, R, and $\mathscr{R}$ are *generalized domains*.[13] The corresponding attributes for general mappings are `Source`, `Range`, and `UnderlyingRelation`.

For each $s \in S$, the set $\{r \in R \mid (s,r) \in \mathscr{R}\}$ is called the set of **images** of s. Analogously, for $r \in R$, the set $\{s \in S \mid (s,r) \in \mathscr{R}\}$ is called the set of **preimages** of r.

The **ordering** of general mappings via $<$ is defined by the ordering of source, range, and underlying relation. Specifically, if the `Source` and `Range` domains of *map1* and *map2* are the same, then one considers the union of the preimages of *map1* and *map2* as a strictly ordered set. The smaller of *map1* and *map2* is the one whose image is smaller on the first point of this sequence where they differ.

For mappings which preserve an algebraic structure, a **kernel** is defined. The operation to compute this kernel is called differently, depending on the structure preserved.[14]

The following is a list of commands for creating general mappings and functions, as well as some important special cases.[15]

- `GeneralMappingByElements( S, R, elms )`
  is the general mapping with source `S` and range `R`, and with underlying relation consisting of the tuples collection `elms`.
- `MappingByFunction( S, R, fun )`
- `MappingByFunction( S, R, fun, invfun )`
  returns a mapping `map` with source `S` and range `R`, such that each element `s` of `S` is mapped to the element `fun( s )`, where `fun` is already a GAP function.

[12] See also the GAP Manual [3], Chapter 31.
[13] *Ibid.*, sec. 12.4.
[14] *Ibid.*, sec. 31.6. Some technical details of general mappings are described in sec. 31.12.
[15] *Ibid.*, sec. 31.1.

If the argument `invfun` is bound, then the resulting `map` is a bijection between `S` and `R`, and the preimage of each element `r` of `R` is given by `invfun( r )`, where `invfun` is a GAP function.

- `InverseGeneralMapping( map )`
  The inverse of a general mapping `map` is the general mapping whose underlying relation contains a pair (r, s) if and only if the underlying relation of `map` contains the pair (s, r).

  Note that the inverse general mapping of `map` is, in general, only a general mapping. If `map` is known to be bijective, then its inverse general mapping will be known to be a mapping. In this case, the command `Inverse( map )` also works.
- `ZeroMapping( S, R )`
  A zero mapping is a total general mapping that maps each element of its source to the zero element of its range. (Each mapping with empty source is a zero mapping.)
- `IdentityMapping( D )`
  is the bijective mapping with source and range equal to the collection `D`, which maps each element of `D` to itself.
- `Embedding( S, T )`
- `Embedding( S, i )`
  returns the embedding of the domain `S` in the domain `T`, or in the second form, some domain indexed by the positive integer `i`. The precise natures of the various methods are described elsewhere.[16]
- `Projection( S, T )`
- `Projection( S, i )`
- `Projection( S )`
  returns the projection of the domain `S` onto the domain `T`, or in the second form, some domain indexed by the positive integer `i`, or in the third form some natural subdomain of `S`. Various methods are defined, and the precise natures of the various methods are described elsewhere.[17]
- `RestrictedMapping( map, subdom )`
  If `subdom` is a subdomain of the source of the general mapping `map`, this operation returns the restriction of `map` to `subdom`.

6.1. Properties and Attributes of (General) Mappings.

- `IsTotal(` m `)`
  is true if each element in the source S of the general mapping m has images – i.e., $s^m \neq \emptyset$, for all $s \in S$ – and false otherwise.

[16] In the GAP Manual: for Lie algebras, see 61.1.3; for group products, see 47.6; for a general description, or for examples see 47.1 for direct products, 47.2 for semidirect products, or 47.4 for wreath products; or for magma rings see 63.3.
[17] *Loc. cit.*

- `IsSingleValued( ` m `)`
  is true if each element in the source S of the general mapping m has at most one image – i.e., $|s^m| \leqslant 1$, for all $s \in S$ – and false otherwise.
- `IsMapping( ` m `)`
  A **mapping** m is a general mapping that assigns to each element x of its source a unique element `Image( ` m, x `)` of its range.

  Equivalently, the general mapping m is a mapping if and only if it is total and single-valued.
- `IsInjective( ` m `)`
  is true if the images of different elements in the source S of the general mapping m are disjoint – i.e., $x^m \cap y^m = \emptyset$, for $x \neq y \in S$ – and false otherwise.
- `IsSurjective( ` m `)`
  is true if each element in the range R of the general mapping m has preimages in the source S of m – i.e., $\{s \in S \mid x \in s^m\} \neq \emptyset$, for all $x \in R$ – and false otherwise.
- `IsBijective( ` m `)`
  A general mapping m is bijective if and only if it is an injective and surjective mapping.
- `Range( ` m `)`
- `Source( ` m `)`
- `UnderlyingRelation( ` m `)`
  The underlying relation of a general mapping m is the domain of pairs (s, r), with s in the source and r in the range of m, and $r \in$ `ImagesElm( ` m, s `)`.
- `UnderlyingGeneralMapping( ` $\mathscr{R}$ `)`
  attribute for underlying relations of general mappings

6.2. **Images under Mappings.**

- `ImagesSource( ` m `)`
  is the set of images of the source of the general mapping m. ImagesSource delegates to ImagesSet, it is introduced only to store the image of m as attribute value.
- `ImagesRepresentative( ` m`, ` x `)`
  If x is an element of the source of the general mapping m then ImagesRepresentative returns either a representative of the set of images of x under m or fail, the latter if and only if x has no images under m. Anything may happen if x is not an element of the source of m.
- `ImagesElm( ` m`, ` x `)`
  If x is an element of the source of the general mapping m then `ImagesElm` returns the set of all images of x under m.

  Anything may happen if x is not an element of the source of m.
- `ImagesSet( ` m`, ` X `)`
  If X is a subset of the source of the general mapping m then ImagesSet returns the set of all images of X under m.

Anything may happen if X is not a subset of the source of m.

- `ImageElm( `m`, `x` )`

  If x is an element of the source of the total and single-valued mapping m then ImageElm returns the unique image of x under map. Anything may happen if x is not an element of the source of map.
- `Image( `m` )`
- `Image( `m`, `x` )`
- `Image( `m`, `X` )`

  `Image( `m` )` is the image of the general mapping m, i.e., the subset of elements of the range of m that are actually values of map. Note that in this case the argument may also be multi-valued.

  `Image( `m`, `x` )` is the image of the element x of the source of the mapping m under m, i.e., the unique element of the range to which m maps x. This can also be expressed as x `^` m. Note that m must be total and single valued, a multi-valued general mapping is not allowed.

  `Image( `m`, `X` )` is the image under m of the subset X of the source of the mapping m; i.e., the subset of the range to which m maps elements of X. Here, X may be a proper set or a domain. The result will be either a proper set or a domain. In this case m may also be multi-valued. (If X and the result are lists then the positions of entries do not, in general, correspond.) `Image` delegates to `ImagesSource` when called with one argument, and to `ImageElm` resp. `ImagesSet` when called with two arguments. If the second argument is not an element or a subset of the source of the first argument, an error is signalled.
- `Images( `m` )`
- `Images( `m`, `x` )`
- `Images( `m`, `X` )`

  `Images( `m` )` is the image of the general mapping m, i.e., the subset of elements of the range of m that are actually values of m.

  `Images( `m`, `x` )` is the set of images of the element x of the source of the general mapping m under m, i.e., the set of elements of the range to which m maps x.

  `Images( `m`, `X` )` is the set of images of the subset X of the source of the general mapping m under m, i.e., the subset of the range to which m maps elements of X . X may be a proper set or a domain. The result will be either a proper set or a domain. (If X and the result are lists then the positions of entries do in general not correspond.) `Images` delegates (to `ImagesSource`, `ImagesElm`, and `ImagesSet`) in the same way that `Image` delegates.

7. Groups Products[18]

This section describes the various group product constructions that are possible in GAP.

For some group products methods are available only if both factors are given in the same representation or only for certain types of groups, such as permutation groups and pc groups, when the product can be naturally represented as a group of the same kind.

In general, GAP does not guarantee that a product of two groups will be in a particular representation.[19] However, GAP will try to choose an efficient representation, so products of permutation groups or pc groups often will be represented as a group of the same kind again. Therefore, the only guaranteed way to relate a product to its factors is via the embedding and projection homomorphisms (sec. 7.2 below).

7.1. **Direct Products.** The direct product of groups is the Cartesian product of the groups (considered as element sets) with component-wise multiplication.

- `DirectProduct( G, H )`
- `DirectProductOp( list, expl )`

  These functions construct the direct product of the groups given as arguments. `DirectProduct` takes an arbitrary positive number of arguments and calls the operation `DirectProductOp`, which takes exactly two arguments, namely a nonempty list of groups and one of these groups. (This somewhat strange syntax allows the method selection to choose a reasonable method for special cases, e.g., if all groups are permutation groups or pc groups.)

  GAP will try to choose an efficient representation for the direct product. For example the direct product of permutation groups will be a permutation group again and the direct product of pc groups will be a pc group.

  If the groups are in different representations a generic direct product will be formed which may not be particularly efficient for many calculations. Instead it may be worth to convert all factors to a common representation first, before forming the product.

  For a product `P` the operation `Embedding( P, nr )` returns the homomorphism embedding the `nr`-th factor into `P`. The operation `Projection( P, nr )` gives the projection of `P` onto the `nr`-th factor (sec. 7.2).

Examples: some basic direct products and their Hasse diagrams.

(1) Let `s3 :=` S_3 and `a3 :=` A_3, the symmetric and alternating groups on three letters, resp.

```
gap> s3 := SymmetricGroup( 3 );;  a3 := AlternatingGroup( 3 );;
```

Of course, S_3 has 6 elements, $\{e, (2,3), (1,2), (1,2,3), (1,3,2), (1,3)\}$, and is generated by $(1,2)$ and $(1,2,3)$; i.e., $S_3 = \langle (1,2), (1,2,3) \rangle$. Its normal subgroup A_3 has three elements

[18] See also the GAP Manual [3], Chapter 47.

[19] Exceptions are `WreathProductImprimitiveAction` and `WreathProductProductAction` which are constructions that make sense only for permutation groups.[20]

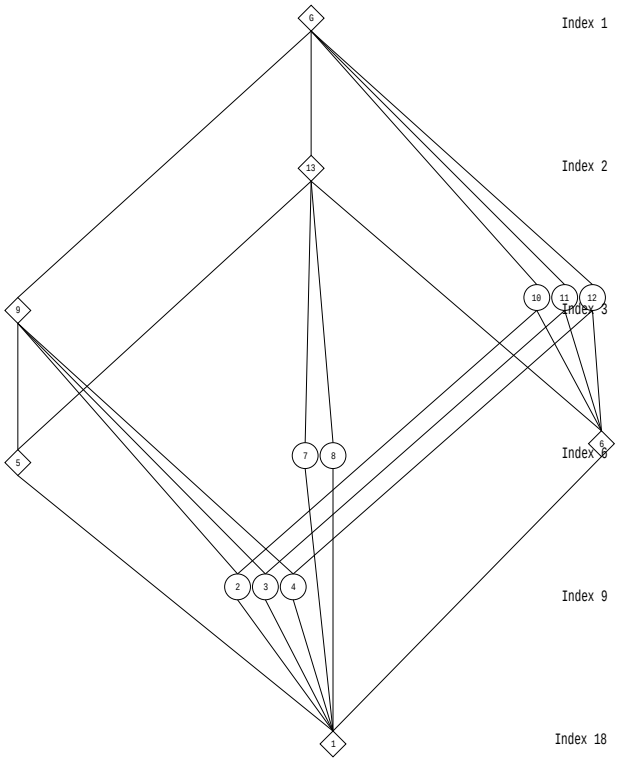


FIGURE 3. Hasse diagram of Sub$[S_3 \times A_3]$ drawn by the XGap program.

$\{e, (1,2,3), (1,3,2)\}$, and $A_3 = \langle (1,2,3) \rangle$. Thus, we could have defined `s3` and `a3` in GAP as follows:

```
gap> s3 := Group( (1,2), (1,2,3) );;  a3 := Group( (1,2,3) );;
```

Now define the direct product group `s3a3 :=` $S_3 \times A_3$, which has $|S_3| \cdot |A_3| = 18$ elements.

```
gap> s3a3 := DirectProduct( s3, a3 );  # returns Group([ (1,2), (1,2,3), (4,5,6) ])
gap> Order( s3a3 );                    # returns 18
```

(If we had defined `a3s3 := DirectProduct( a3, s3 );`, the result would have been `Group([ (1,2,3), (4,5), (4,5,6) ])`, which is, of course, isomorphic to `s3a3`.)

If you are using the XGap package, an extension of GAP, you can see the Hasse diagram of the subgroup lattice of a group with the command `GraphicSubgroupLattice`. For example, the following command draws the subgroup lattice of $S_3 \times A_3$ (Figure 3):[21]

```
gap> GraphicSubgroupLattice( s3a3 );
```

[21]When you execute the `GraphicSubgroupLattice` command, the initial result is a diagram showing only the trivial subgroups (e.g., (e) and $S_3 \times A_3$). To see the full subgroup lattice, you must select `All Subgroups` from the `Subgroups` menu.

XGap depicts normal subgroups with diamonds and non-normal subgroups with circles. Conjugacy classes of subgroups are grouped together. (Thus, a normal subgroup appears by itself.)

In the lattice in Figure 3, the subgroup of index two is the normal subgroup $A_3 \times A_3$. Now, A_3 has no proper nontrivial subgroups (and thus $\mathrm{Sub}[A_3]$ is just the two element chain). Nonetheless, it is clear from the diagram that the subgroup lattice $\mathrm{Sub}[A_3 \times A_3]$ is isomorphic to M_4 (cf. $\mathrm{Sub}[\mathbb{Z}_2 \times \mathbb{Z}_2] \cong M_3$).

Remark: $\mathrm{Sub}[A_3 \times A_3] \cong M_4 \cong \mathrm{Sub}[S_3]$. Indeed,

- It is well-known that $\mathrm{Sub}[G] \cong M_4$ iff G is (isomorphic to) $C_3 \times C_3$ or D_3.
- S_3 is isomorphic to the dihedral group D_3 of order 6, the symmetries of an equilateral triangle (three reflections and three rotations; some authors refer to D_3 as D_6.) We can easily check by hand that $S_3 \cong D_3$, but GAP quickly confirms this fact as follows:

```
gap> s3 := SymmetricGroup(3);    # returns Sym( [ 1 .. 3 ] )
gap> d3 := DihedralGroup(6);     # returns <pc group of size 6 with 2 generators>
gap> IsDihedralGroup(s3);        # returns true
```

- A_3 is isomorphic to C_3. This is obvious, but let's check it anyway using GAP:

```
gap> a3 := AlternatingGroup(3);  # returns Alt( [ 1 .. 3 ] )
gap> Elements(a3);               # returns [ (), (1,2,3), (1,3,2) ]
gap> IsCyclic(a3);               # returns true
```

(2) For our next example, we let `a3 :=` A_3 and `a4 :=` A_4, the alternating groups on three and four letters (resp.), and let `c2 :=` $\mathbb{Z}_2$, the cyclic group or order 2.

```
gap> a3 := AlternatingGroup( 3 );;  a4 := AlternatingGroup( 4 );;
gap> c2 := CyclicGroup(2);
<pc group of size 2 with 1 generators>

gap> a4c2 := DirectProduct( a4, c2 );
<group of size 24 with 3 generators>

gap> a4a3 := DirectProduct( a4, a3 );
Group([ (1,2,3), (2,3,4), (5,6,7) ])
```

If you are using XGap, the subgroup lattices $\mathrm{Sub}[A_4 \times A_3]$ and $\mathrm{Sub}[A_4 \times \mathbb{Z}_2]$ are displayed with `GraphicSubgroupLattice`.

```
gap> GraphicSubgroupLattice( a4a3 );
gap> GraphicSubgroupLattice( a4c2 );
```

In the subgroup lattice $\mathrm{Sub}[A_4 \times \mathbb{Z}_2]$ (Figure 5), consider the (maximal) normal subgroup of index 3; i.e., the diamond labelled 24. The diagram makes clear that this is the subgroup $V_4 \times \mathbb{Z}_2$, where V_4 is the Klein 4 group. It's hard to see exactly what's happening below this subgroup in the full $\mathrm{Sub}[A_4 \times \mathbb{Z}_2]$ lattice, but we can get a handle on $V_4 \times \mathbb{Z}_2$, and draw its subgroup lattice, $\mathrm{Sub}[V_4 \times \mathbb{Z}_2]$ (Figure 6), with the following commands:

```
gap> cclsa4 := ConjugacyClassesSubgroups(a4);
[ Group( () )^G, Group( [ (1,2)(3,4) ] )^G, Group( [ (2,4,3) ] )^G,
  Group( [ (1,3)(2,4), (1,2)(3,4) ] )^G,
```

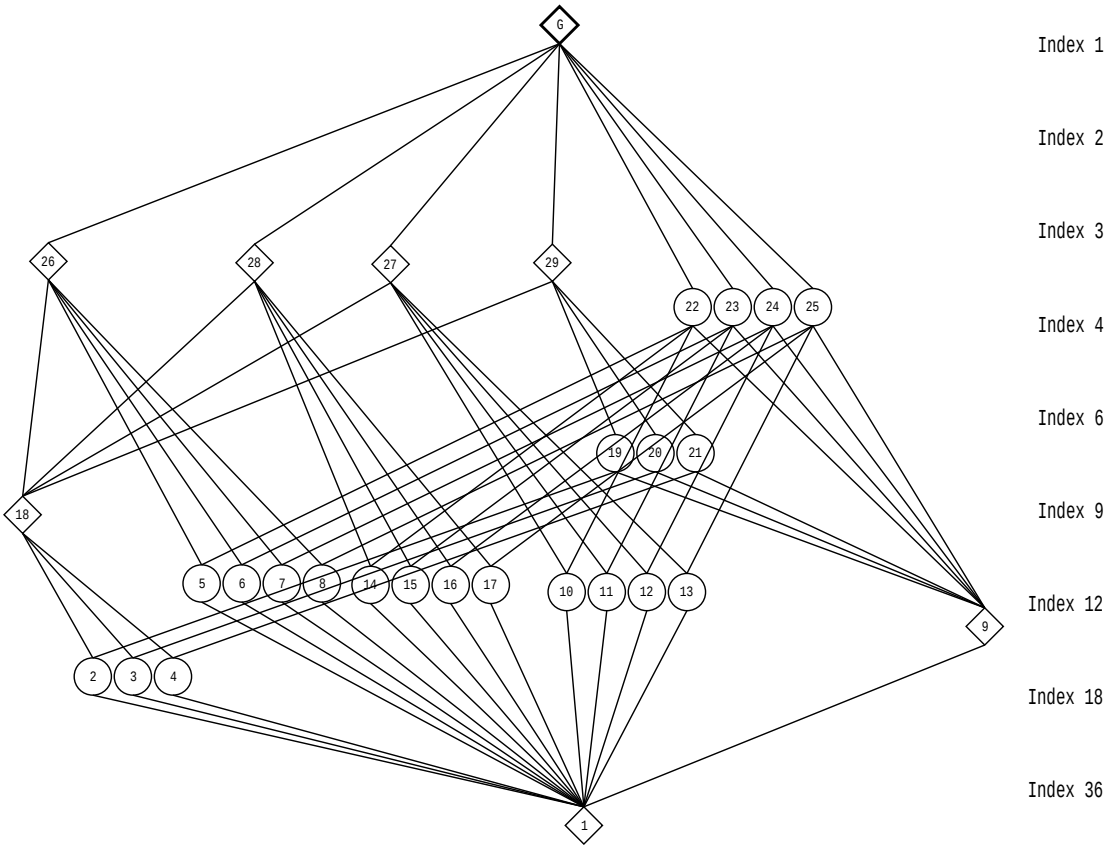


FIGURE 4. Hasse diagram of Sub$[A_4 \times A_3]$ drawn by the XGap program.

```
  Group( [ (1,3)(2,4), (1,2)(3,4), (2,4,3) ] )^G ]
gap> v4 := Representative(cclsa4[4]);
Group([ (1,3)(2,4), (1,2)(3,4) ])
gap> Order(v4);     # returns 4
gap> IsCyclic(v4);  # returns false (v4 is, indeed, the Klein 4 group)
gap> v4c2 := DirectProduct(v4,c2);
<group of size 8 with 3 generators>
gap> GraphicSubgroupLattice(v4c2);
```

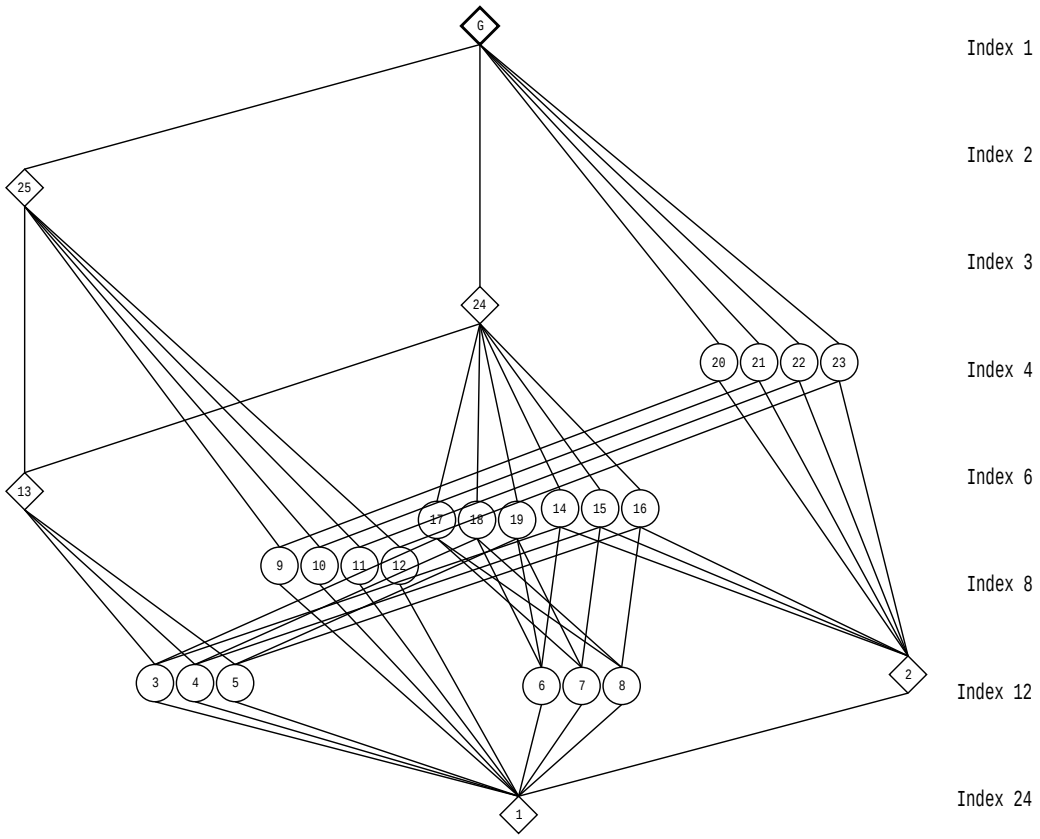


FIGURE 5. Hasse diagram of Sub$[A_4 \times \mathbb{Z}_2]$ drawn by the XGap program.

7.2. **Embeddings and Projections.** The relation between a group product and its factors is provided via homomorphisms, the embeddings in the product and the projections from the product. Depending on the kind of product only some of these are defined.

- `Embedding( P, nr )`

  returns the `nr`-th embedding in the group product `P`. The actual meaning of this embedding is described in the section for the appropriate product.
- `Projection( P[, nr] )`

  returns the (`nr`-th) projection of the group product `P`. The actual meaning of the projection returned is described in the section for the appropriate product.

Examples: embeddings and projections of $S_3 \times A_3$.

As above, let `s3 :=` S_3 and `a3 :=` A_3, the symmetric and alternating group on three letters (resp.), and let `s3a3 :=` $S_3 \times A_3$.

```
gap> s3 := Group( (1,2), (1,2,3) );;  a3 := Group( (1,2,3) );;
gap> s3a3 := DirectProduct( s3, a3 );  # returns Group([ (1,2), (1,2,3), (4,5,6) ])
```

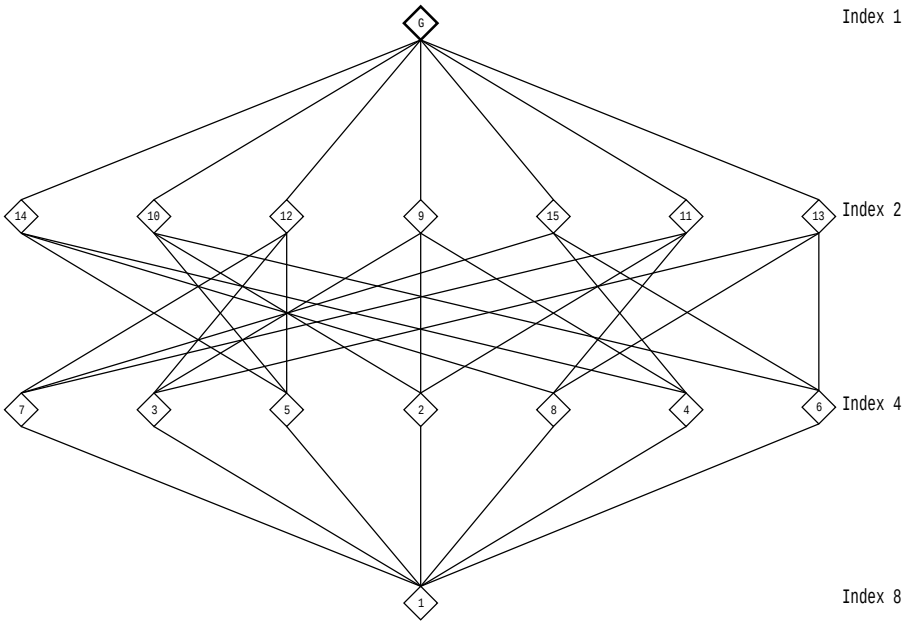


FIGURE 6. Hasse diagram of Sub$[V_4 \times \mathbb{Z}_2]$ drawn by the XGap program.

```
gap> Order( s3a3 );                    # returns  18
```

The following GAP code illustrates the behavior of the commands `Embedding` and `Projection`, and the corresponding `Source`, `Range`, and `Image` commands (sec. 6):

```
gap> i1 := Embedding( s3a3, 1 );
1st embedding into Group([ (1,2), (1,2,3), (4,5,6) ])
gap> i2 := Embedding( s3a3, 2 );
2nd embedding into Group([ (1,2), (1,2,3), (4,5,6) ])

gap> Source( i1 );                    # returns Group([ (1,2), (1,2,3) ])
gap> Range( i1 );                     # returns Group([ (1,2), (1,2,3), (4,5,6) ])
gap> Image( i1 );                     # returns Group([ (1,2), (1,2,3) ])
gap> ImagesSource( i1 );              # returns Group([ (1,2), (1,2,3) ])

gap> Source( i2 );                    # returns Group([ (1,2,3) ])
gap> Range( i2 );                     # returns Group([ (1,2), (1,2,3), (4,5,6) ])
gap> Image( i2 );                     # returns Group([ (4,5,6) ])
gap> ImagesSource( i2 );              # returns Group([ (4,5,6) ])

gap> p1 := Projection( s3a3, 1 );
1st projection of Group([ (1,2), (1,2,3), (4,5,6) ])
gap> p2 := Projection( s3a3, 2 );
```

```
2nd projection of Group([ (1,2), (1,2,3), (4,5,6) ])

gap> Source( p1 );                  # returns Group([ (1,2), (1,2,3), (4,5,6) ])
gap> Range( p1 );                   # returns Group([ (1,2), (1,2,3) ])
gap> Image( p1 );                   # returns Group([ (1,2), (1,2,3) ])
gap> Image( p1, (1,2,3) );          # returns (1,2,3)
gap> Image( p1, (4,5,6) );          # returns ()
gap> Image( p1, (1,2)(4,5,6) );     # returns (1,2)

gap> Source( p2 );                  # returns Group([ (1,2), (1,2,3), (4,5,6) ])
gap> Range( p2 );                   # returns Group([ (1,2,3) ])
gap> Image( p2 );                   # returns Group([ (1,2,3) ])
gap> Image( p2, (1,2,3) );          # returns ()
gap> Image( p2, (4,5,6) );          # returns (1,2,3)
gap> Image( p2, (1,2)(4,5,6) );     # returns (1,2,3)

gap> Image( p2, (1,2,3)(4,5) );     # Error: (1,2,3)(4,5) is not in Source of p2
brk> quit;

gap> Elements( Source( p2 ) );
[ (), (4,5,6), (4,6,5), (2,3), (2,3)(4,5,6), (2,3)(4,6,5), (1,2),
  (1,2)(4,5,6), (1,2)(4,6,5), (1,2,3), (1,2,3)(4,5,6), (1,2,3)(4,6,5),
  (1,3,2), (1,3,2)(4,5,6), (1,3,2)(4,6,5), (1,3), (1,3)(4,5,6), (1,3)(4,6,5) ]

gap> Image( p2, (1,2,3)(4,6,5) );   # returns (1,3,2)
```

We could have used the command `UnderlyingRelation` (sec. 6.1) to display most of this information at once:

```
gap> Elements(UnderlyingRelation( i1 ));
[ Tuple( [ (), () ] ), Tuple( [ (2,3), (2,3) ] ), Tuple( [ (1,2), (1,2) ] ),
  Tuple( [ (1,2,3), (1,2,3) ] ), Tuple( [ (1,3,2), (1,3,2) ] ),
  Tuple( [ (1,3), (1,3) ] ) ]

gap> Elements(UnderlyingRelation( i2 ));
[ Tuple( [ (), () ] ), Tuple( [ (1,2,3), (4,5,6) ] ),
  Tuple( [ (1,3,2), (4,6,5) ] ) ]

gap> Elements(UnderlyingRelation( p1 ));
[ Tuple( [ (), () ] ), Tuple( [ (4,5,6), () ] ), Tuple( [ (4,6,5), () ] ),
  Tuple( [ (2,3), (2,3) ] ), Tuple( [ (2,3)(4,5,6), (2,3) ] ),
  Tuple( [ (2,3)(4,6,5), (2,3) ] ), Tuple( [ (1,2), (1,2) ] ),
  Tuple( [ (1,2)(4,5,6), (1,2) ] ), Tuple( [ (1,2)(4,6,5), (1,2) ] ),
  Tuple( [ (1,2,3), (1,2,3) ] ), Tuple( [ (1,2,3)(4,5,6), (1,2,3) ] ),
  Tuple( [ (1,2,3)(4,6,5), (1,2,3) ] ), Tuple( [ (1,3,2), (1,3,2) ] ),
  Tuple( [ (1,3,2)(4,5,6), (1,3,2) ] ), Tuple( [ (1,3,2)(4,6,5), (1,3,2) ] ),
  Tuple( [ (1,3), (1,3) ] ), Tuple( [ (1,3)(4,5,6), (1,3) ] ),
  Tuple( [ (1,3)(4,6,5), (1,3) ] ) ]
```

```
gap> Elements(UnderlyingRelation( p2 ));
[ Tuple( [ (), () ] ), Tuple( [ (4,5,6), (1,2,3) ] ),
  Tuple( [ (4,6,5), (1,3,2) ] ), Tuple( [ (2,3), () ] ),
  Tuple( [ (2,3)(4,5,6), (1,2,3) ] ), Tuple( [ (2,3)(4,6,5), (1,3,2) ] ),
  Tuple( [ (1,2), () ] ), Tuple( [ (1,2)(4,5,6), (1,2,3) ] ),
  Tuple( [ (1,2)(4,6,5), (1,3,2) ] ), Tuple( [ (1,2,3), () ] ),
  Tuple( [ (1,2,3)(4,5,6), (1,2,3) ] ), Tuple( [ (1,2,3)(4,6,5), (1,3,2) ] ),
  Tuple( [ (1,3,2), () ] ), Tuple( [ (1,3,2)(4,5,6), (1,2,3) ] ),
  Tuple( [ (1,3,2)(4,6,5), (1,3,2) ] ), Tuple( [ (1,3), () ] ),
  Tuple( [ (1,3)(4,5,6), (1,2,3) ] ), Tuple( [ (1,3)(4,6,5), (1,3,2) ] ) ]
gap>
```

But note that `UnderlyingRelation` does not tell the whole story. In particular, it is impossible to infer from the output of `UnderlyingRelation` the full `Source` and `Range` of a general mapping.

7.3. **Semidirect Products.** The semidirect product of a group N with a group G acting on N via a homomorphism α from G into the automorphism group of N is the Cartesian product $G \times N$ with the multiplication

$$(g, n) \cdot (h, m) = (gh, n\bar{h} \cdot m) = (gh, n^{h\alpha}m).$$

- `SemidirectProduct(` G, α, N `)`
- `SemidirectProduct(` *autgp*, N `)`

  constructs the semidirect product of N with G acting via α, where α must be a homomorphism from G into a group of automorphisms of N.

  If N is a full row space over a field $\mathbb{F}$, α must be a homomorphism from G into a matrix group of the right dimension over a subfield of $\mathbb{F}$, or into a permutation group (in this case permutation matrices are taken).

  In the second variant, *autgp* must be a group of automorphism of N, it is a shorthand for `SemidirectProduct(` *autgp*`, IdentityMapping(`*autgp*`),` N `)`. Note that (unless *autgrp* has been obtained by the operation `AutomorphismGroup`) you have to test `IsGroupOfAutomorphisms(` *autgrp* `)` to ensure that GAP knows that *autgrp* consists of group automorphisms.

```
gap> n:=AbelianGroup(IsPcGroup,[5,5]);
<pc group of size 25 with 2 generators>
gap> au:=DerivedSubgroup(AutomorphismGroup(n));;
gap> Size(au);                      # returns  120
gap> p:=SemidirectProduct(au,n);
<permutation group with 5 generators>
gap> Size(p);                       # returns 3000
gap> n:=Group((1,2),(3,4));;
gap> au:=AutomorphismGroup(n);;
gap> au:=First(Elements(au),i->Order(i)=3);;
gap> au:=Group(au);
<group with 1 generators>
gap> SemidirectProduct(au,n);
Error, no method found! For debugging hints type ?Recovery from NoMethodFound
```

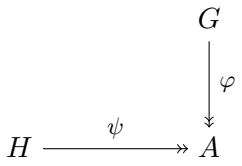
```
Error, no 2nd choice method found for 'IsomorphismPcGroup' on 1 arguments
gap> IsGroupOfAutomorphisms(au);
true
gap> SemidirectProduct(au,n);
<pc group with 3 generators>
gap> n:=AbelianGroup(IsPcGroup,[2,2]);
<pc group of size 4 with 2 generators>
gap> au:=AutomorphismGroup(n);
<group of size 6 with 2 generators>
gap> apc:=IsomorphismPcGroup(au);
CompositionMapping( Pcgs([ (2,3), (1,2,3) ]) ->
[ f1, f2 ], <action isomorphism> )
gap> g:=Image(apc);
Group([ f1, f2 ])
gap> apci:=InverseGeneralMapping(apc);
[ f1*f2^2, f1*f2 ] -> [ Pcgs([ f1, f2 ]) -> [ f1*f2, f2 ],
Pcgs([ f1, f2 ]) -> [ f2, f1 ] ]
gap> IsGroupHomomorphism(apci);
true
gap> p:=SemidirectProduct(g,apci,n);
<pc group of size 24 with 4 generators>
gap> IsomorphismGroups(p,Group((1,2,3,4),(1,2)));
[ f1, f2, f3, f4 ] -> [ (2,3), (2,3,4), (1,4)(2,3), (1,2)(3,4) ]
gap> SemidirectProduct(SU(3,3),GF(9)^3);
<matrix group of size 4408992 with 3 generators>
gap> SemidirectProduct(Group((1,2,3),(2,3,4)),GF(5)^4);
<matrix group of size 7500 with 3 generators>
gap> g:=Group((3,4,5),(1,2,3));;
gap> mats:=[[[Z(2^2),0*Z(2)],[0*Z(2),Z(2^2)^2]],
> [[Z(2)^0,Z(2)^0], [Z(2)^0,0*Z(2)]]];;
gap> hom:=GroupHomomorphismByImages(g,Group(mats),[g.1,g.2],mats);;
gap> SemidirectProduct(g,hom,GF(4)^2);
<matrix group of size 960 with 3 generators>
gap> SemidirectProduct(g,hom,GF(16)^2);
<matrix group of size 15360 with 4 generators>
```

For the semidirect product P of G with N, `Embedding( P, 1 )` embeds G, `Embedding( P, 2 )` embeds N. The operation `Projection(P)` returns the projection of P onto G.[22]

```
gap> Size(Image(Embedding(p,1)));    # returns 6
gap> Embedding(p,2);
[ f1, f2 ] -> [ f3, f4 ]
gap> Projection(p);
[ f1, f2, f3, f4 ] -> [ f1, f2, <identity> of ..., <identity> of ... ]
```

[22] *Op. cit.*, sec. 47.6.

7.4. **Subdirect Products**[23]**.** The subdirect product of the groups G and H with respect to the epimorphisms $\varphi : G \to A$ and $\psi : H \to A$ (for a common group A) is the subgroup of the direct product $G \times H$ consisting of the elements (g, h) for which $\varphi(g) = \psi(h)$. It is the pull-back of the diagram:



- `SubdirectProduct(` G `,` H`,` φ`,` ψ `)`
  constructs the subdirect product of G and H with respect to the epimorphisms φ from G onto a group A and ψ from H onto the same group A. For a subdirect product `P`, the operation `Projection(P, nr)` returns the projection on the `nr`-th factor. (In general the factors do not embed in a subdirect product.)

```
gap> g:=Group((1,2,3),(1,2));;
gap> hom:=GroupHomomorphismByImagesNC(g,g,[(1,2,3),(1,2)],[(),(1,2)]);
[ (1,2,3), (1,2) ] -> [ (), (1,2) ]
gap> s:=SubdirectProduct(g,g,hom,hom);
Group([ (1,2,3), (1,2)(4,5), (4,5,6) ])
gap> Size(s);                                  # returns 18
gap> p:=Projection(s,2);
2nd projection of Group([ (1,2,3), (1,2)(4,5), (4,5,6) ])
gap> Image(p,(1,3,2)(4,5,6));                  # returns (1,2,3)
```

- `SubdirectProducts(` G`,` H `)`
  computes all subdirect products of G and H up to conjugacy in Parent(G) $\times$ Parent(H). The subdirect products are returned as subgroups of this direct product.

7.5. **Wreath Products**[24]**.** The *wreath product* of a group G with a permutation group P acting on n points is the semidirect product of the normal subgroup G^n with the group P which acts on G^n by permuting the components.

- `WreathProduct(` G`,` P `)`
- `WreathProduct(` G`,` H `[,` *hom*`]` `)`
  constructs the wreath product of the group G with the permutation group P (acting on its `MovedPoints`).

  The second usage constructs the wreath product of the group G with the image of the group H under *hom* where *hom* must be a homomorphism from H into a permutation group. If *hom* is not given, and P is not a permutation group, then *hom* is taken to be the result of `IsomorphismPermGroup(P)` (whose degree may be dependent on the method and thus is

[22] *Ibid.*, sec. 47.3.
[23] *Ibid.*, sec. 47.4.

not well-defined!).

If W is a wreath product of G with a permutation group P of degree n, and $1 \leqslant nr \leqslant n$, then the operation `Embedding( W, nr )` provides the embedding of G in the nr-th component of the direct product of the base group G^n of W. `Embedding( W, n+1 )` is the embedding of P in W. The operation `Projection( W )` gives the projection onto the acting group P.

```
gap> g:=Group((1,2,3),(1,2));     # the symmetric group on {1, 2, 3}
gap> p:=Group((1,2,3));           # the alternating group on {1, 2, 3}
gap> w:=WreathProduct(g,p);
Group([ (1,2,3), (1,2), (4,5,6), (4,5), (7,8,9), (7,8), (1,4,7)(2,5,8)(3,6,9) ])
gap> Order(w);                    # returns 648
```

The wreath product `w` is the semidirect product $S_3 \times S_3 \times S_3 \rtimes A_3$, which of course has order $|S_3|^3 \cdot |A_3| = 6^3 \cdot 6 = 648$.

```
gap> Embedding(w,1);
1st embedding into Group( [ (1,2,3), (1,2), (4,5,6), (4,5), (7,8,9), (7,8),
(1,4,7)(2,5,8)(3,6,9) ] )
gap> Image(Embedding(w,3));     # returns Group([ (7,8,9), (7,8) ])
gap> Image(Embedding(w,4));     # returns Group([ (1,4,7)(2,5,8)(3,6,9) ])
gap> Image(Projection(w),(1,4,8,2,6,7,3,5,9));  # returns (1,2,3)
```

- `WreathProductImprimitiveAction( G, H )`
  for two permutation groups G and H this function constructs the wreath product of G and H in the imprimitive action. If G acts on l points and H on m points this action will be on l^m points, it will be imprimitive with m blocks of size l each. The operations `Embedding` and `Projection` operate on this product as described for general wreath products.

  ```
  gap> w:=WreathProductImprimitiveAction(g,p);;
  gap> LargestMovedPoint(w);      # returns 9
  ```

- `WreathProductProductAction( G, H )`
  for two permutation groups G and H this function constructs the wreath product in product action. If G acts on l points and H on m points this action will be on l^m points.

  The operations `Embedding` and `Projection` operate on this product as described for general wreath products.

  ```
  gap> w:=WreathProductProductAction(g,p);
  <permutation group of size 648 with 7 generators>
  gap> LargestMovedPoint(w);      # returns 27
  ```

8. Matrix Groups and the Classical Groups

8.1. **Fields.** This subsection contains a highly abridged version of chapters 56 and 57 of the GAP manual. Initially, my main goal was to collect the few things we'll need to compute with matrix groups in GAP, but because the subject of fields is so important and interesting, I've included a bit more information here than is required for working with matrix groups.

A *division ring* is a ring in which every non-zero element has an inverse. The most important class of division rings are the commutative ones, which are called *fields*.

If a field F is a subfield of a commutative ring C, C can be considered as a vector space over the (left) acting domain F. In this situation, we call F the *field of definition* of C. Each field in GAP is represented as a vector space over a subfield, thus each field is in fact a field extension in a natural way, which is used by functions such as `Norm` and `Trace`.

- `IsDivisionRing(` D `)`
  A *division ring* is a nontrivial associative algebra D in which each nonzero element has a multiplicative inverse. In GAP every division ring is a vector space over a division ring (possibly over itself). Note that being a division ring is not a property that a ring can acquire, because a ring is usually not represented as a vector space. The field of coefficients is stored as `LeftActingDomain(` D `)`.
- `IsField( D )`
  A *field* is a commutative division ring.

```
gap> IsField( GaloisField(16) );          # returns true (the field with 16 elements)
gap> IsField( Rationals );                # returns true (the field of rationals)
gap> q:= QuaternionAlgebra( Rationals );; # (a noncommutative division ring)
gap> IsField( q );  IsDivisionRing( q );  # returns false true
gap> mat:= [ [ 1 ] ];; a:= Algebra( Rationals, [ mat ] );;
gap> IsDivisionRing( a );                 # returns false  (since the algebra a
                                          # was not constructed as a division ring)
```

- `Field(` $z, \ldots$ `)`
- `Field( list )`
- `Field(` F`, list )`
  returns the smallest field K that contains all the elements $z, \ldots$, or (resp.) the smallest field K that contains all elements in `list`. If no subfield F is given, K is constructed as a field over itself. In the third form, `Field` constructs the field generated by the field F and the elements in the list `list`, as a vector space over F.
- `DefaultField(` $z, \ldots$ `)`
- `DefaultField( list )`
  returns a field K that contains all the elements $z, \ldots$, or (resp.) a field K that contains all elements in the list `list`. The result need not be the smallest field in which the elements lie, in contrast to the field returned by the `Field` command. For example, for elements from cyclotomic fields, `DefaultField` returns the smallest cyclotomic field in which the elements lie, but the elements may lie in a smaller number field which is not a cyclotomic field.

- `Z(p,d)`
- `Z(p^d)`

  For creating elements of a finite field the function `Z` can be used. The call `Z(p,d)` (alternatively `Z(p^d)`) returns the designated generator of the multiplicative group of the finite field with p^d elements. Here p must be a prime.

  GAP can represent elements of all finite fields `GF(p^d)` such that either (1) `p^d <= 65536` (in which case an extremely efficient internal representation is used); (2) `d = 1`, (in which case, for large `p`, the field is represented using the machinery of Residue Class Rings (see section 14.4 of the GAP manual) or (3) if the Conway Polynomial of degree `d` over `GF(p)` is known, or can be computed. If you attempt to construct an element of `GF(p^d)` for which `d > 1` and the relevant Conway Polynomial is not known, and not necessarily easy to find (see 57.5.2), then GAP will stop with an error and enter the break loop. If you leave this break loop by entering return; GAP will attempt to compute the Conway Polynomial, which may take a very long time.

  The root returned by `Z` is a generator of the multiplicative group of the finite field with pd elements, which is cyclic. The order of the element is of course $p^d - 1$. The $p^d - 1$ different powers of the root are exactly the nonzero elements of the finite field. Thus all nonzero elements of the finite field with p^d elements can be entered as `Z(p^d)^i`. Note that this is also the form that GAP uses to output those elements when they are stored in the internal representation. In larger fields, it is more convenient to enter and print elements as linear combinations of powers of the primitive element. (See section 57.6 of the GAP manual.)

  The additive neutral element is `0*Z(p)`. The multiplicative neutral element is `Z(p)^0`. It is different from the integer 1, not only in the obvious way – e.g., `Z(2)^0 + Z(2)^0` is `0*Z(2)`, while `1+1` is `2` – but also in more subtle ways.[25]

```
# some finite fields
gap> Z(2);                          # returns Z(2)^0
gap> Field( Z(2) );                 # returns GF(2)
gap> Elements( Field( Z(2) ) );     # returns [ 0*Z(2), Z(2)^0 ]

gap> Z(5);                          # returns Z(5)
gap> Field( Z(5) );                 # returns GF(5)
gap> Elements( Field( Z(5) ) );     # returns [ 0*Z(5), Z(5)^0, Z(5), Z(5)^2, Z(5)^3 ]

gap> Z(32);                         # returns Z(2^5)
gap> Field( Z(32) );                # returns GF(2^5)

gap> Z(2)+Z(2); Z(2)*Z(2);          # returns 0*Z(2)  Z(2)^0
gap> Z(5)+Z(5); Z(5)*Z(5);          # returns Z(5)^2  Z(5)^2
gap> Z(32)+Z(32); Z(32)*Z(32);      # returns 0*Z(2)  Z(2^5)^2

gap> Field(Z(4)); Field(Z(8));      # returns GF(2^2)  GF(2^3)
gap> Elements( Field( Z(4) ) );     # returns [ 0*Z(2), Z(2)^0, Z(2^2), Z(2^2)^2 ]
gap> Elements( Field( Z(8) ) );     # returns [ 0*Z(2), Z(2)^0, Z(2^3), Z(2^3)^2, Z(2^3)^3, ...
                                                        ..., Z(2^3)^4, Z(2^3)^5, Z(2^3)^6 ]
gap> Field( [ Z(4), Z(8) ] );       # returns GF(2^6) (the field of 64 elements)
```

[25] See section 57.1 of the GAP manual [3].

Since finite field elements are scalars, the operations `Characteristic, One, Zero, Inverse, AdditiveInverse, Order` can be applied to them.[26]

```
gap> Characteristic( Z( 16 )^10 );       # returns 2
gap> Characteristic( Z( 9 )^2 );         # returns 3
gap> Characteristic( [ Z(4), Z(8) ] );   # returns 2
gap> One( Z(9) );                        # returns Z(3)^0
gap> One( 0*Z(4) );                      # returns Z(2)^0
gap> Zero(Z(125));                       # returns 0*Z(5)
gap> Inverse( Z(9) );                    # returns Z(3^2)^7
gap> AdditiveInverse( Z(9) );            # returns Z(3^2)^5
gap> Order( Z(9)^7 );                    # returns 8
```

8.2. **Matrix groups.**[27] Matrix groups are groups generated by invertible square matrices. For example,

```
gap> m1 := [ [ Z(3)^0, Z(3)^0, Z(3) ],
>            [ Z(3), 0*Z(3), Z(3) ],
>            [ 0*Z(3), Z(3), 0*Z(3) ] ];;
gap> m2 := [ [ Z(3), Z(3), Z(3)^0 ],
>            [ Z(3), 0*Z(3), Z(3) ],
>            [ Z(3)^0, 0*Z(3), Z(3) ] ];;
gap> m := Group( m1, m2 );
Group(
[ [ [ Z(3)^0, Z(3)^0, Z(3) ], [ Z(3), 0*Z(3), Z(3) ], [ 0*Z(3), Z(3), 0*Z(3) ] ],
[ [ Z(3), Z(3), Z(3)^0 ], [ Z(3), 0*Z(3), Z(3) ], [ Z(3)^0, 0*Z(3), Z(3) ] ] ])
```

Some attributes and operations available for matrix groups are the following:

- `IsMatrixGroup( grp )`

  For most operations, GAP only provides methods for finite matrix groups. Many calculations in finite matrix groups are done via a NiceMonomorphism (see 38.5) that represents a faithful action on vectors.
- `DimensionOfMatrixGroup( matgrp )`
  returns the dimension of the matrix group `matgrp`.
- `DefaultFieldOfMatrixGroup( matgrp )`
  A returns a field containing all the matrix entries of all elements of `matgrp`. It is not guaranteed to be the smallest field with this property.
- `FieldOfMatrixGroup( matgrp )`
  returns the smallest field containing all the matrix entries of all elements of `matgrp`. As the calculation of this can be hard, this should onlybe used if one really needs the smallest field, use `DefaultFieldOfMatrixGroup` to get (for example) the characteristic.

[26] For a full description of these operations, see the Gap manual [3] sec. 30.10.
[27] See Chapter 42 of the GAP manual [3] for more details.

- `TransposedMatrixGroup( matgrp )`
  returns the transpose of the matrix group `matgrp`. The transpose of the transpose of `matgrp` is identical to `matgrp`.

```
gap> DimensionOfMatrixGroup(m);       # returns 3
gap> DefaultFieldOfMatrixGroup(m);    # returns GF(3)

gap> G := Group( [[0,-1],[1,0]] );;
gap> IsMatrixGroup(G);                # returns true
gap> T := TransposedMatrixGroup( G ); # returns Group([ [ [ 0, 1 ], [ -1, 0 ] ] ])
gap> IsIdenticalObj( G, TransposedMatrixGroup( T ) );  # returns true
```

8.3. **Classical groups.** The *general linear group*, $\mathrm{GL}(n, R)$, is the group of all invertible $n \times n$ matrices over a ring R. The *special linear group*, $\mathrm{SL}(n, R)$, is the group of all invertible $n \times n$ matrices over a ring R whose determinant is 1.

- `IsGeneralLinearGroup( grp )`
- `IsGL( grp )`
  tests whether a group is isomorphic to a general linear group;
- `IsNaturalGL( matgrp )`
  tests whether a matrix group is the general linear group in the right dimension over the (smallest) ring which contains all entries of its elements.

  Analogous methods are available for special linear groups.

```
# Let m and G be the groups defined above.
gap> IsMatrixGroup(m); IsGL(m);              # returns true false
gap> IsMatrixGroup(G); IsGL(G);              # returns true false

gap> gl := GL(2,3);                          # returns GL(2,3)
gap> sl := SL(3,2);                          # returns SL(3,2)
gap> IsMatrixGroup(gl); IsGL(gl); IsSL(gl);  # returns true true false
gap> IsMatrixGroup(sl); IsGL(sl); IsSL(sl);  # returns true true true
```

- `GeneralLinearGroup( [ filt, ] ` d, R `)`
- `GL( [ filt, ] ` d, R `)`
- `GeneralLinearGroup( [ filt, ] ` d, q `)`
- `GL( [ filt, ] ` d, q `)`
  The first two forms construct a group isomorphic to the general linear group $GL(d, R)$ of all $d \times d$ matrices that are invertible over the ring R, in the category given by the filter `filt`. The third and the fourth form construct the general linear group over the finite field with q elements. If `filt` is not given it defaults to `IsMatrixGroup`, and the returned group is the general linear group as a matrix group in its natural action (see also 42.3.2, 42.5.4). Currently supported rings `R` are finite fields, the ring `Integers`, and residue class rings `Integers mod` m.

```
gap> GL(4,3);                # returns GL(4,3)
gap> GL(2,Integers);         # returns GL(2,Integers)
gap> GL(3,Integers mod 12);  # returns GL(3,Z/12Z)
```

- `SpecialLinearGroup( [filt, ] ` d, R `)`

- `SL( [filt, ]` d, R `)`
- `SpecialLinearGroup( [filt, ]` d, q `)`
- `SL( [filt, ]` d, q `)`

  The first two forms construct a group isomorphic to the special linear group $SL(d, R)$ of all those $d \times d$ matrices over the ring R whose determinant is the identity of R, in the category given by the filter `filt`. The third and the fourth form construct the special linear group over the finite field with q elements. If `filt` is not given it defaults to `IsMatrixGroup`, and the returned group is the special linear group as a matrix group in its natural action (see also 42.3.4, 42.5.5). Currently supported rings R are finite fields, the ring `Integers`, and residue class rings `Integers mod` m.

```
gap> SpecialLinearGroup(2,2);  # returns SL(2,2)
gap> SL(3,Integers);           # returns SL(3,Integers)
gap> SL(4,Integers mod 4);     # returns SL(4,Z/4Z)
```

  Using the `OnLines` operation it is possible to obtain the corresponding projective groups in a permutation action:

```
gap> g:=GL(4,3);;Size(g);      # returns 24261120
gap> pgl:=Action(g,Orbit(g,Z(3)^0*[1,0,0,0],OnLines),OnLines);;
gap> Size(pgl);                # returns 12130560
```

8.4. **Conjugacy classes in classical groups.** For general and special linear groups, GAP has an efficient method to generate representatives of the conjugacy classes. This uses results from linear algebra on normal forms of matrices. [28]

```
gap> g := SL(4,9);              # returns SL(4,9)
gap> NrConjugacyClasses(g);     # returns 861
gap> cl := ConjugacyClasses(g);;
gap> Length(cl);                # returns 861
```

- `NrConjugacyClassesGL( n, q )`
- `NrConjugacyClassesSL( n, q )`

  for given integer n and prime power q, computes the number of conjugacy classes in the classical groups $\mathrm{GL}(n, q)$ and $\mathrm{SL}(n, q)$, resp.[29] Analogous operations are available for the other classical groups, $\mathrm{GU}(n, q)$, $\mathrm{SU}(n, q)$, $\mathrm{PGL}(n, q)$, $\mathrm{PGU}(n, q)$, $\mathrm{PSL}(n, q)$, and $\mathrm{PSU}(n, q)$.

[28] If you know how to do this for other types of classical groups, the GAP folks would like to hear from you.

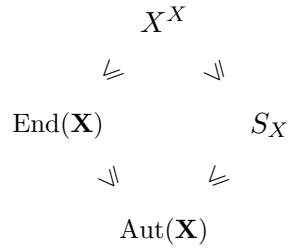
[29] See also sections 37.10.2 and 48.2 of the GAP manual [3].

9. Representations

In the first subsection below, we review some very basic theory of representations of finite groups. If this is too familiar, skip to the subsection 9.2 where we give some examples demonstrating how to use GAP to work with group actions and permutation representations.

9.1. **Permutation representations of finite groups.** Let X be a finite set and consider the set X^X of all maps from X to itself, which, when endowed with composition of maps and the identity mapping, forms a monoid, $\langle X^X; \circ, \mathrm{id}_X \rangle$. The submonoid S_X of all bijective maps in X^X is a group, the *symmetric group on* X. When the underlying set isn't important, we write S_n to denote the generic symmetric group on an n-element set.

If we have defined some set F of basic operations on X, so that $\mathbf{X} = \langle X; F \rangle$ is an algebra, then two other important submonoids of X^X are $\mathrm{End}(\mathbf{X})$, the set of maps in X^X which respect all operations in F, and $\mathrm{Aut}(\mathbf{X})$, the set of bijective maps in X^X which respect all operations in F. It is apparent from the definition that $\mathrm{Aut}(\mathbf{X}) = S_X \cap \mathrm{End}(\mathbf{X})$, and $\mathrm{Aut}(\mathbf{X})$ is a submonoid of $\mathrm{End}(\mathbf{X})$ and a subgroup of S_X. These four fundamental monoids associated with the algebra $\mathbf{X}$ are shown in the diagram below.



Given a finite group G, and an algebra $\mathbf{X} = \langle X; F \rangle$, a *representation* of G on $\mathbf{X}$ is a group homomorphism from G into $\mathrm{Aut}(\mathbf{X})$. That is, a representation of G is a mapping $\varphi : G \to \mathrm{Aut}(\mathbf{X})$ which satisfies $\varphi(g_1 g_2) = \varphi(g_1) \circ \varphi(g_2)$, where (as above) $\circ$ denotes composition of maps in $\mathrm{Aut}(\mathbf{X})$.

Thus, a representation defines an action by G on the set X: $\bar{g}x = \varphi(g)(x)$. If $\bar{G} = \varphi[G]$ denotes the image of G under φ, then $\langle X; \bar{G} \rangle$ is a G-set.[30] The action is called *transitive* iff for each pair $x, y \in X$ there is some $g \in G$ such that $\varphi(g)(x) = y$. The representation φ is called *faithful* iff it is a monomorphism, in which case G is isomorphic to its image under φ, which is a subgroup of $\mathrm{Aut}(\mathbf{X})$. We also say, in this case, that the group acts faithfully, and call it a *permutation group*. A group which acts transitively on some set is called a *transitive group*. Without specifying the set, however, this term is meaningless, since every group acts transitively on some sets and intransitively on others. A representation φ is called transitive iff the resulting action is transitive.

Two special cases are almost always what one means when one speaks of a representation of a finite group. They are the so called

- *linear representations*, where $\mathbf{X} = \langle X; +, \circ, -, 0, 1, \mathbb{F} \rangle$ is a finite dimensional vector space over a field $\mathbb{F}$, so $\mathrm{Aut}(\mathbf{X})$ is the set of invertible matrices with entries from $\mathbb{F}$;
- *permutation representations*, where $\mathbf{X} = X$ is just a set, so $\mathrm{Aut}(\mathbf{X}) = S_X$.

[30] More generally, a G-set is sometimes defined to be a pair (X, φ), where φ is a homomorphism from a group into the symmetric group S_X; see e.g. [5].

The following elementary theorem tells us precisely when a particular group G has a transitive permutation representation on a set of size n. The theorem is easy to prove.[31]

Theorem 1. *Let G be a group. The following three conditions are equivalent.*

(i) There is a transitive permutation representation of G on a set of size n.

(ii) There is a homomorphism from G into S_n such that the image of G is transitive.

(iii) The group G has a subgroup of index n.

For a given group G, and any subgroup $H < G$, we define a transitive permutation representation of G, which we denote ρ_H. Specifically, ρ_H is a group homomorphism from G into the symmetric group $\mathrm{Sym}(G/H)$ of permutations on the set $G/H = \{H, Hx_1, Hx_2, \dots\}$ of *right* cosets of H in G. The action is simply right-multiplication by elements of G. That is:

$$\rho_H : G \to \mathrm{Sym}(G/H), \quad \text{where} \quad \rho_H(g)(Hx) = Hxg.$$

With this set-up, to check the homomorphism property of ρ_H, we should write the permutation mappings in $\mathrm{Sym}(G/H)$ on the right of their arguments, as in $Hx\rho_H(g) = Hxg$. For then we have

$$Hx\rho_H(g_1g_2) = Hx(g_1g_2) = Hxg_1g_2 = Hx\rho_H(g_1)\rho_H(g_2);$$

i.e. $\rho_H(g_1g_2) = \rho_H(g_1)\rho_H(g_2)$.

For each $Hx \in G/H$, the *point stabilizer* of Hx is

$$G_{Hx} = \{g \in G : Hxg = Hx\} = \{g \in G : Hxgx^{-1} = H\} = x^{-1}G_Hx = x^{-1}Hx = H^x,$$

so the kernel of the homomorphism ρ_H is

$$\ker \rho_H = \{g \in G : \forall x \in G,\ Hxg = Hx\} = \bigcap_{x \in G} x^{-1}Hx = \bigcap_{x \in G} H^x.$$

Note that $\ker \rho_H$ is the largest normal subgroup of G contained in H, also known as the *core of H in G*, which we denote by

$$\mathrm{Core}_G(H) = \bigcap_{x \in G} H^x.$$

Next we describe (up to equivalence) all transitive permutation representations of a given group G. We call two representations (or actions) *equivalent* iff the associated G-sets are isomorphic. The foregoing implies that every transitive permutation representation of G is equivalent to ρ_H for some subgroup $H < G$. The following lemma[32] shows that we need only consider a single representative H from each of the conjugacy classes of subgroups.

Lemma 1. *Suppose G acts transitively on two sets, A and B. Fix $a \in A$ and let G_a be the stabilizer of a (under the first action). Then the two actions are equivalent if and only if the subgroup G_a is also a stabilizer under the second action of some point $b \in B$.*

The point stabilizers of the action ρ_H described above are the conjugates of H in G. Therefore, the lemma implies that, for any two subgroups $H, K \leqslant G$, the representations ρ_H and ρ_K

[31] See, e.g., [5] Theorem 7.16.

[32] Lemma 1.6B of [1].

are equivalent precisely when $K = x^{-1}Hx$ for some $x \in G$. Hence, the transitive permutation representations of G are given, up to equivalence, by ρ_{K_i} as K_i runs over a set of representatives of conjugacy classes of subgroups of G.

9.2. **Example: the transitive permutation representations of A_5.** GAP has a function for determining the conjugacy classes of subgroups, which we now use to find (up to equivalence) all permutation representations of the group A_5. First, we define the alternating group on five points, and then compute the conjugacy classes of subgroups.[33]

```
gap> a5 := AlternatingGroup(5);;
gap> ccls := ConjugacyClassesSubgroups( a5 );
[ Group( () )^G, Group( [ (2,3)(4,5) ] )^G, Group( [ (3,4,5) ] )^G,
  Group( [ (2,3)(4,5), (2,4)(3,5) ] )^G, Group( [ (1,2,3,4,5) ] )^G,
  Group( [ (3,4,5), (1,2)(4,5) ] )^G, Group( [ (1,2,3,4,5), (2,5)(3,4) ] )^G,
  Group( [ (2,3)(4,5), (2,4)(3,5), (3,4,5) ] )^G, AlternatingGroup( [ 1 .. 5 ] )^G ]
```

If you are running XGap, an extension of GAP, you can see a diagram of the entire subgroup lattice of a group (of reasonably small order). For example, at the `xgap` command line we could type `GraphicSubgroupLattice( a5 )`. This opens a new window showing just the two subgroups (e) and A_5. Selecting `All Subgroups` from the `Subgroups` drop-down menu draws a (rather messy) Hasse diagram of Sub$[A_5]$. You can then move around the various conjugacy classes of subgroups (which stayed glued together) to get a pretty good picture of Sub$[A_5]$. (See Figure 7.)

If we use the XGap program as described above we could count the conjugacy classes of subgroups by looking at the Hasse diagram of Sub$[A_5]$. However, it's more convenient (and faster) to simply do:

```
gap> ccsg := ConjugacyClassesSubgroups( a5 );;
gap> Size(ccsg);
```

which returns 9, telling us that A_5 has 9 conjugacy classes of subgroups. This includes the singleton classes $\{(e)\}$ and $\{A_5\}$, which GAP calls `Group( () )^G` and `AlternatingGroup( [ 1 .. 5 ] )^G`, respectively. Let's examine the other seven classes. We get a list of representative subgroups, one for each class, as follows:

```
gap> clreps := List( ccsg, x -> Representative( x ));
[ Group(()), Group([ (2,3)(4,5) ]), Group([ (3,4,5) ]), Group([ (2,3)(4,5), (2,4)(3,5) ]),
  Group([ (1,2,3,4,5) ]), Group([ (3,4,5), (1,2)(4,5) ]),
  Group([ (1,2,3,4,5), (2,5)(3,4) ]), Group([ (2,3)(4,5), (2,4)(3,5), (3,4,5) ]), Alt( [ 1 .. 5 ] ) ]
```

The orders and indices of these subgroups are given by

```
gap> List( clreps, x -> Order(x));         % returns [ 1, 2, 3, 4, 5, 6, 10, 12, 60 ]
gap> List( clreps, x -> Index( a5, x ) );  % returns [ 60, 30, 20, 15, 12, 10, 6, 5, 1 ]
```

[33]Note: `ConjugacyClassesSubgroups` does not compute the ordinary conjugacy classes of elements of the group. (Those are found with the command `ConjugacyClasses`.) Rather, `ConjugacyClassesSubgroups( G )` partitions the set Sub$[G]$ of subgroups of G into conjugacy classes *of subgroups*.

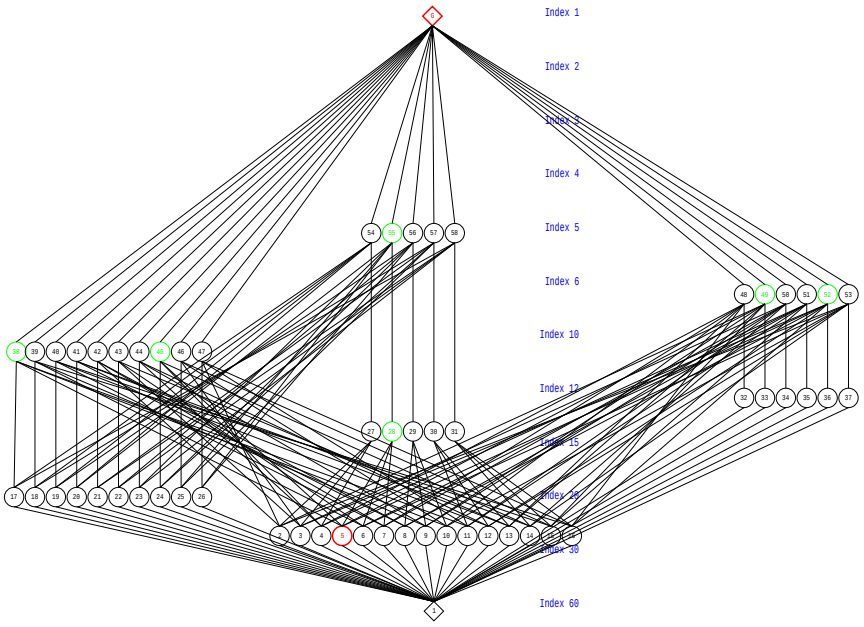


FIGURE 7. Hasse diagram of Sub$[A_5]$ drawn by the XGap program. Colored green are the subgroups in the interval above one of the $\mathbb{Z}_2$ subgroups of A_5. Thus, A_5 acts transitively on the 30 cosets of $\mathbb{Z}_2$, and the permutational algebra $\langle A_5/\mathbb{Z}_2; A_5\rangle$ has congruence lattice isomorphic to the interval $[\mathbb{Z}_2, A_5]$.

(Of course, any subgroup $H \leqslant A_5$ has order $|H| = |A_5|\,[A_5 : H] = 60\,[A_5 : H]$.) From the list of subgroup orders, we see that `clreps[2]`, `clreps[3]`, and `clreps[5]` must be the groups $\mathbb{Z}_2$, $\mathbb{Z}_3$, and $\mathbb{Z}_5$ (the only groups of orders two, three, and five, respectively). We can easily identify the other subgroups as well.[34] For example, `clreps[4]` has order 4, so it must be either $\mathbb{Z}_2 \oplus \mathbb{Z}_2$ or $\mathbb{Z}_4$. Deciding to which isomorphism class `clreps[4]` belongs is simply a matter of checking whether it's cyclic. These and other subgroups can be determined using the following GAP commands:

```
gap> IsCyclic( clreps[4] );            # returns false
gap> IsAbelian( clreps[6] );           # returns false
gap> IsDihedralGroup( clreps[6] );     # returns true
gap> IsDihedralGroup( clreps[7] );     # returns true
gap> IsAlternatingGroup( clreps[8] );  # returns true
```

Therefore, `clreps[4]` must be the Klein four group $\mathbb{Z}_2 \oplus \mathbb{Z}_2$, `clreps[6]` must be D_3, `clreps[7]` must be D_5. Finally, `clreps[8]` has order 12, so it must be one of $\mathbb{Z}_2 \oplus \mathbb{Z}_6$, $\mathbb{Z}_{12}$, A_4, D_6, or the group T (generated by two elements a, b where $|a| = 6, b^2 = a^3$, and $ba = a^{-1}b$). The last command above shows that `clreps[8]` is A_4.

[34] A nice reference list of all groups of orders 1 through 15 is given on pp. 98–9 of Hungerford [4].

The foregoing demonstrates some useful GAP commands, but we could have identified all these subgroups in one step with the `StructureDescription` command:

```
gap> List(clreps, x->StructureDescription(x));
[ "1", "C2", "C3", "C2 x C2", "C5", "S3", "D10", "A4", "A5" ]
```

Now, the representations of A_5 are all faithful since A_5 is simple. Below is a table of all seven (equivalence classes of) permutation representations of A_5 on cosets A_5/H, ordered by the number of such cosets (i.e. the index $[A_5 : H]$):

Conj. class rep. H	Index $[A_5 : H]$	Representation homomorphism	Is it primitive?
A_4	5	$\rho_{A_4} : A_5 \hookrightarrow \mathrm{Sym}(A_5/A_4) \cong S_5$	yes
D_5	6	$\rho_{D_5} : A_5 \hookrightarrow \mathrm{Sym}(A_5/D_5) \cong S_6$	yes
D_3	10	$\rho_{D_3} : A_5 \hookrightarrow \mathrm{Sym}(A_5/D_3) \cong S_{10}$	yes
$\mathbb{Z}_5$	12	$\rho_{\mathbb{Z}_5} : A_5 \hookrightarrow \mathrm{Sym}(A_5/\mathbb{Z}_5) \cong S_{12}$	no
V_4	15	$\rho_{V_4} : A_5 \hookrightarrow \mathrm{Sym}(A_5/V_4) \cong S_{15}$	no
$\mathbb{Z}_3$	20	$\rho_{\mathbb{Z}_3} : A_5 \hookrightarrow \mathrm{Sym}(A_5/\mathbb{Z}_3) \cong S_{20}$	no
$\mathbb{Z}_2$	30	$\rho_{\mathbb{Z}_2} : A_5 \hookrightarrow \mathrm{Sym}(A_5/\mathbb{Z}_2) \cong S_{30}$	no
(e)	60	$\rho : A_5 \hookrightarrow \mathrm{Sym}(A_5) \cong S_{60}$	no

We can use GAP to verify that A_5 does indeed show up as a transitive subgroup of some of the symmetric groups listed in the table above. For example, the first two are checked as follows:

```
gap> NrTransitiveGroups(5);  % returns 5
gap> NrTransitiveGroups(6);  % returns 16
gap> List([1..5], x->StructureDescription(TransitiveGroup(5,x)));
[ "C5", "D10", "C5 : C4", "A5", "S5" ]
gap> List([1..16], x->StructureDescription(TransitiveGroup(6,x)));
[ "C6", "S3", "D12", "A4", "C3 x S3", "C2 x A4", "S4", "S4", "S3 x S3", "(C3 x C3) : C4",
  "C2 x S4", "A5", "(S3 x S3) : C2", "S5", "A6", "S6" ]
```

The last line above indicates that (some copy of) A_5 shows up as a transitive subgroup of S_6. (Of course, it is *not* the copy of $A_5 < S_6$ that moves only five points!)

The last column of the table above was filled in simply by looking at the subgroup lattice of A_5 in Figure 7. In general, if H is a coatom in $\mathrm{Sub}[G]$ (i.e. a maximal subgroup of G), then the representation

$$\rho_H : G \to \mathrm{Sym}(G/H) \cong S_{[G:H]}$$

is primitive.

G-sets. Defining G-sets with GAP is very useful and important. It allows us to work with and analyze a particular permutation representation. Let us consider the A_5-set given by A_5 acting on cosets of $H = \mathrm{clreps}[2] = C_2$. In GAP we do

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

```
gap> ccsg := ConjugacyClassesSubgroups( a5 );;
gap> H := Representative( ccsg[3] );                    # C3
gap> Gbar := Action( G, RightCosets(G,H), OnRight );;  # a subgroup of S20 isomorphic to A5
gap> MovedPoints( Gbar );                               # [ 1, ..., 20 ]
gap> AllBlocks( Gbar );                    # [ [ 1, 2, 3, 4 ], [ 1, 6, 11, 16 ], [ 1, 20 ] ]
gap> for b in AllBlocks( Gbar ) do Print( Orbit( Gbar, b, OnSets ), "\n"); od;
[ [ 1, 2, 3, 4 ], [ 17, 18, 19, 20 ], [ 13, 14, 15, 16 ], [ 9, 10, 11, 12 ], [ 5, 6, 7, 8 ] ]
[ [ 1, 6, 11, 16 ], [ 2, 7, 12, 17 ], [ 3, 8, 13, 18 ], [ 4, 9, 14, 19 ], [ 5, 10, 15, 20 ] ]
[ [ 1, 20 ], [ 16, 17 ], [ 12, 13 ], [ 11, 18 ], [ 8, 9 ], [ 7, 14 ], [ 6, 19 ], [ 4, 5 ], [ 3, 10 ], [ 2, 15 ] ]
```

The command `MovedPoints` shows that `Gbar` $\cong A_5$ acts transitively on the set $G/H = A_5/Z_3$ of 30 cosets. (We could also have checked that the action is transitive using `Orbits(Gbar)` and noting that there is just one orbit.) The command `AllBlocks(Gbar)` shows the first block of each nontrivial "system of imprimitivity" (or congruence) of the G-set $\langle G/H, \bar{G}\rangle$. Finally, the last command displays the three nontrivial congruences, and shows $\mathrm{Con}\langle G/H, \bar{G}\rangle \cong M_3$. Another way to see that $\mathrm{Con}\langle G/H, \bar{G}\rangle \cong M_3$ is to check the sublattice of intermediate subgroups between H and G, as follows:

```
gap> intHG := IntermediateSubgroups(G, H);
rec( subgroups := [ Group([ (3,4,5), (1,2)(4,5) ]), Group([ (3,4,5), (2,3)(4,5) ]), Group([ (3,4,5), (1,5,3) ]) ],
  inclusions := [ [ 0, 1 ], [ 0, 2 ], [ 0, 3 ], [ 1, 4 ], [ 2, 4 ], [ 3, 4 ] ] )
```

This results in an object, which I've called `intHG`, having fields `intHG.subgroups` and `intHG.inclusions`. The inclusions field shows the covering relations in the sublattice interval $[H, G] \leqslant \mathrm{Sub}[G]$.

9.3. **Example: A_8 factor groups, natural homs, and representations of normalizers.**[35] The group generated by the permutations (1,2) and (1,2,3,4,5,6,7,8) is S_8, the symmetric group on eight points. We assign it to the identifier s8 as follows:

```
gap> s8 := Group( (1,2), (1,2,3,4,5,6,7,8) );;
```

Now S_8 contains the alternating group on eight points which can be described in several ways, e.g., as the group of all even permutations in S_8, or as the derived subgroup of S_8.

```
gap> a8 := DerivedSubgroup( s8 );
Group([ (1,2,3), (2,3,4), (2,4)(3,5), (2,6,4), (2,4)(5,7), (2,8,6,4)(3,5) ])

gap> Size( a8 );                                  % returns 20160
gap> IsAbelian( a8 );                             % returns false
gap> IsPerfect( a8 );                             % returns true
gap> syl2 := SylowSubgroup( a8, 2 );;
gap> Size( syl2 );                                % returns 64
gap> Normalizer( a8, syl2 ) = syl2;               % returns true
                                                  % (Sylow subgroups are self-normalizng.)
gap> cent := Centralizer( a8, Centre( syl2 ) );;
gap> Size( cent );                                % returns 192
```

[35] From chapter 5 of the GAP tutorial [2].

```
gap> DerivedSeries( cent );; List( last, Size );  % returns [ 192, 96, 32, 2, 1 ]
```

Next we want to calculate a subgroup of $G =$ `a8`, then its normalizer, and finally the structure of the extension. We begin by forming a subgroup generated by three commuting involutions, i.e., a subgroup isomorphic to the additive group of the vector space 2^3. (We will sometimes refer to this subgroup as $H =$ `elab`.)

```
gap> elab := Group( (1,2)(3,4)(5,6)(7,8), (1,3)(2,4)(5,7)(6,8), (1,5)(2,6)(3,7)(4,8) );;
gap> Size( elab );                                % returns 8
gap> IsElementaryAbelian( elab );                 % returns true
```

As usual, GAP prints the group by giving all its generators. This can be annoying, especially if there are many of them or if they are of huge degree. You can give a name to the group itself using the function `SetName`. We do this with the name `2^3` below which reflects the mathematical properties of the group. From now on, GAP will use this name when printing the group for us, but we cannot use this name to specify the group to GAP, because the name does not know to which group it was assigned. When talking to the computer, you must always use identifiers.

```
gap> SetName( elab, "2^3" ); elab;                    % returns 2^3
gap> norm := Normalizer( a8, elab );; Size( norm );   % returns 1344
```

Now that we have the subgroup $N_G(H) =$ `norm` of order 1344 and its subgroup $H =$ `elab`, we want to look at its factor group $N_G(H)/H$. We also want to find preimages of factor group elements, so we use the **natural homomorphism** defined on $N_G(H) =$ `norm` with kernel $H =$ `elab` and image $N_G(H)/H$.

```
gap> hom := NaturalHomomorphismByNormalSubgroup( norm, elab );
<action epimorphism>
gap> f := Image( hom );
Group([ (), (), (), (4,5)(6,7), (4,6)(5,7), (2,3)(6,7), (2,4)(3,5),(1,2)(5,6) ])
gap> Size( f );                           % returns 168
```

The factor group `f =` $N_G(H)/H =$ `norm/elab` is again represented as a permutation group. (However, the action domain of this factor group has nothing to do with the action domain of `norm`.) We can now form images and preimages under the natural homomorphism. The set of preimages of an element under `hom` is a coset modulo $H =$ `elab`.

```
gap> ker:= Kernel( hom );                     % returns  2^3
gap> x := (1,8,3,5,7,6,2);; Image( hom, x );  % returns  (1,7,5,6,2,3,4)
gap> coset := PreImages( hom, last );         % returns  RightCoset(2^3,(2,8,6,7,3,4,5))
```

Note that GAP is free to choose any representative for the coset of preimages. Of course, if x and y are two representatives, the quotient $x^{-1}y$ lies in the kernel $H =$ `ker` of the homomorphism.

```
gap> rep:= Representative( coset );           % returns  (2,8,6,7,3,4,5)
gap> x * rep^-1 in ker;                       % returns  true
```

The factor group $N_G(H)/H =$ `f` is a *simple* group; i.e., it has no non-trivial normal subgroups. GAP can detect this fact, and it can then also find the name by which this simple group is known among group theorists. (Such names are of course not available for non-simple groups.)

```
gap> IsSimple( f );                              % returns true
gap> IsomorphismTypeInfoFiniteSimpleGroup( f );
rec( series := "L", parameter := [ 2, 7 ],
name := "A(1,7) = L(2,7) ~ B(1,7) = O(3,7) ~ C(1,7) = S(2,7) ~ 2A(1,7) = U(2,7) ~ A(2,2) = L(3,2)" )
gap> SetName( f, "L_3(2)" );
```

We give `f` the name `L_3(2)` because the last part of the name string reveals that it is isomorphic to the simple linear group $L_3(2)$. This group, however, also has a lot of other names.

The group $N_G(H) =$ `norm` acts on the eight elements of its normal subgroup $H =$ `elab` by conjugation, yielding a representation of $L_3(2)$ in S_8 which leaves one point fixed (namely, the point 1).

More precisely, let $\tau : N_G(H) \to \mathrm{Sym}(H)$ denote the conjugation representation (where $\mathrm{Sym}(H) \cong S_8$, since H has 8 elements). That is, for each $g \in N_G(H)$, $\tau : g \mapsto \tau_g \in \mathrm{Sym}(H)$, where $\tau_g(h) = ghg^{-1}$, for $h \in H$. Now, the kernel

$$\ker \tau = \{g \in N_G(H) \mid ghg^{-1} = h \text{ for all } h \in H\}$$

clearly contains H, since H is abelian. Actually, $\ker \tau$ is equal to H. (Otherwise, $\ker \tau / H$ would be a nontrivial normal subgroup of $N_G(H)/H$, which is impossible, since $N_G(H)/H$ is simple.) Therefore, the image of the representation is $\tau(N_G(H)) \cong N_G(H)/H \cong L_3(2)$. This image can be computed with the function `Action`, which we will use below. But first recall that, from the start, we had

$$2^3 \cong H \trianglelefteq N_G(H) \leqslant A_8 \trianglelefteq S_8$$

and the image $\tau(N_G(H))$ is a subgroup of $\mathrm{Sym}(H) \cong S_8$. In fact, $\tau(N_G(H)) \leqslant N_G(H)$, and we can show that $N_G(H)$ is indeed a split extension of the elementary abelian group H with the image $\tau(N_G(H))$.

```
gap> op := Action( norm, elab );
Group([ (), (), (), (5,6)(7,8), (5,7)(6,8), (3,4)(7,8), (3,5)(4,6), (2,3)(6,7) ])
gap> IsSubgroup( a8, op );                   % returns true
gap> IsSubgroup( norm, op );                 % returns true
gap> IsTrivial( Intersection( elab, op ) );  % returns true
gap> SetName( norm, "2^3:L_3(2)" );
```

By the way, you should not try the operator `<` instead of the function `IsSubgroup`. Something like

```
gap> elab < a8;          % returns false
```

will not cause an error, but the result does not signify anything about the inclusion of one group in another; `<` tests which of the two groups is less in some total order. On the other hand, the equality operator = in fact does test the equality of its arguments.

10. Miscellaneous Useful Commands

Lists. A list is constructed as follows:

```
gap> mylist := [ 2, 3, 5, 7, 11, 13, 17, 19 ];;
```

You can extract certain elements from a list to generate new lists as follows:

```
gap> mylist{[4..6]};     # returns [ 7, 11, 13 ]
gap> mylist{[1,7,1,8]};  # returns [ 2, 17, 2, 19 ]
```

It is possible to nest such sublist extractions, as shown in the following example:

```
gap> m := [ [1,2,3], [4,5,6], [7,8,9], [10,11,12] ];;
gap> m{[1,2,3]}{[3,2]};           # returns [ [ 3, 2 ], [ 6, 5 ], [ 9, 8 ] ]
gap> l := m{[1,2,3]};; l{[3,2]};  # returns [ [ 7, 8, 9 ], [ 4, 5, 6 ] ]
```

Note the difference between the last two outputs.

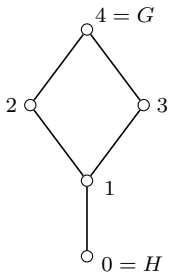
In one of our applications, we constructed a list of lists of intervals in subgroup lattices. Here's a simple example:

```
gap> G:=SymmetricGroup(5);
gap> H:=Representative(ccsg[8]);   # returns  Group([ (1,2,3,4,5) ])
gap> StructureDescription(H);      # returns  "C5"
gap> intHG:=IntermediateSubgroups(G,H);;
```

The function `IntermediateSubgroups(G,H)` returns an object which has two fields: `subgroups` containing a lists of the subgroups *strictly* between H and G, and `inclusions` which contains a list of covering relations among the subgroups in `subgroups`. For example, if `[ 3, 5 ]` appears in `inclusions`, it means the subgroup labelled 3 is a maximal subgroup of the subgroup labelled 5. The number 0 denotes H and the highest number denotes G. Continuing with the example above, we have

```
gap> intHG.inclusions;  # returns [ [ 0, 1 ], [ 1, 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 4 ] ] ]
```

This tells us that the interval from $H = C_5$ up to $G = S_5$ is the lattice of subgroups shown in the figure below.



Now, suppose we want to start a list of such covering relation lists; i.e. a list of intervals in subgroup lattices. We store the first interval in `incl`:

```
gap> incl:=[intHG.inclusions];  # returns [ [ [ 0, 1 ], [ 1, 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 4 ] ] ]
```

We get the next interval we want, say,

```
gap> K:=Representative(ccsg[10]);;
gap> intKG:=IntermediateSubgroups(G,K);;
```

Then we use the `Add` function to append the new list of covering relations to the original list as follows:

```
gap> Add(incl,intKG.inclusions);
gap> incl;
[ [ [ 0, 1 ], [ 1, 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 4 ] ],
  [ [ 0, 1 ], [ 0, 2 ], [ 1, 3 ], [ 2, 3 ] ] ]
```

We may want to search all groups for such upper intervals and store them without repetition. This is a bit trickier because, for example, GAP views the two lists

```
[ [ 0, 1 ], [ 1, 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 4 ] ],
[ [ 0, 1 ], [ 1, 3 ], [ 1, 2 ], [ 3, 4 ], [ 2, 4 ] ]
```

as distinct, though we recognize them as the same lattice (shown in the figure above). This is easily remedied by sorting the lists. The first list is already sorted lexicographically. We sort the second as follows:

```
SSortedList([ [ 0, 1 ], [ 1, 3 ], [ 1, 2 ], [ 3, 4 ], [ 2, 4 ] ]);
[ [ 0, 1 ], [ 1, 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 4 ] ]
```

and see that the sorted version matches the first list. Unfortunately, not all repetitions of lattice intervals can be avoided simply by sorting. Consider, for example, the two hexagons,

```
[ [ 0, 1 ], [ 0, 2 ], [ 1, 3 ], [ 2, 4 ], [ 3, 5 ], [ 4, 5 ] ]
[ [ 0, 1 ], [ 0, 3 ], [ 1, 2 ], [ 2, 5 ], [ 3, 4 ], [ 4, 5 ] ]
```

These are already sorted, yet they are distinct lists which give the same lattice. One solution (perhaps not the most efficient) is given by the following routine, `isIsomorphicInterval`, which checks whether two lists of covering relations represent isomorphic intervals; i.e. whether they are the same modulo a relabelling of the elements.

```
isIsomorphicInterval:=function(list1, list2)
    # Gap function for testing whether two sets of covering relations
    # are the same modulo relabelling.

    local n, m, j, list3, G, p;

    if not IsList(list1) or not IsList(list2) then
        Error("usage: isIsomorphicInterval( <lst1>, <lst2> );");
    fi;
    if Length(list1) <> Length(list2)  then
        return false;
    fi;

    list1 := SSortedList(list1);
    list2 := SSortedList(list2);

    n:=Length(list1);

    m:=Maximum(Maximum(list1));
```

```
    G:=SymmetricGroup(m);

    for p in Elements(G) do
        list3 := [[ ]];
        for j in [1..n] do
            list3[j]:=List(list1[j], x->((x+1)^p)-1);
        od;
        list3:=SSortedList(list3);
        if list3=list2 then
            return true;
        fi;
    od;
    return false;
end;
```

References

[1] John Dixon and Brian Mortimer. *Permutation Groups*. Springer-Verlag, New York, 1996.

[2] The GAP Group. *GAP: A Tutorial*. http://www.gap-system.org, December 2008. Release 4.4.12.

[3] The GAP Group. *GAP: Reference Manual*. http://www.gap-system.org, December 2008. Release 4.4.12.

[4] Thomas W. Hungerford. *Algebra*. Springer-Verlag, New York, 1974.

[5] Michio Suzuki. *Group Theory*, volume 1. Springer-Verlag, New York, 1982.