Worksheet 6

Version 2015-09-14

6 More on groups

6.1 Action and orbits

Let G be a group and X a non-empty set. A right action of G on X is a map

$$\omega : X \times G \to X \ , (x, g) \mapsto x.g$$

such that for all $x \in X$ and $g, h \in G$ we have

$$\omega(x, 1_G) = x.1_G = x \qquad \text{and} \qquad \omega(\omega(x, g), h) = (x.g).h = x.(gh) = \omega(x, gh) \ .$$

For example, if $X = \{1, \ldots, n\}$ and $G \leq \Sigma_n$, then G acts on X from the right via $x.g := x^g$ (i.e. apply the permutation g to x).

For $a \in X$, the orbit of a under G is

$$a.G := \{a.g \mid g \in G\} \ .$$

6.2 Computing orbits: naive approach

Computing orbits is one of the central building blocks of computational group theory. A naive way to compute an orbit is to directly use the definition:

```
# "G" is a group, "a" is some object, and "act" is a
# function which realizes the desired action of the group G.
orbitNaive := function(G, a, act)
    local orb, g, b;
    orb := [];
    for g in G do
        b := act(a, g);
        Add(orb, b);
    od;
    # At this point orb is just a list, and may contain duplicates.
    # An orbit is a set, so let's turn it into one.
    return Set(orb);
end;
```

We can now do this:

```
gap> orbitNaive(SymmetricGroup(5), 1, function(a,g) return a^g; end);
[ 1, 2, 3, 4, 5 ]
```

We could have implemented the orbit computation without the `act` function, and simply written `a^g` directly; but by having `act`, we gain flexibility and can work with arbitrary actions. But since that particular action is quite common, GAP has a shortcut for it: `OnPoints`. Many other typical actions are also provided, e.g. `OnRight`, `OnLeftInverse`, `OnSets`, `OnTuples` and more. As usual, consult the GAP manual for details.

Exercise 6.2.1

For $2 \leq n \leq 10$, let $G := \Sigma_n$ and compute $1.G$ using `orbitNaive`. How long does this take in each case? Also experiment with other groups.

Of course GAP also has its own command for computing orbits.

```
gap> Orbit(SymmetricGroup(5), 1, OnPoints);
[ 1, 2, 3, 4, 5 ]
gap> Orbit(SymmetricGroup(5), 1); # if you omit 'act', GAP uses OnPoints
[ 1, 2, 3, 4, 5 ]
```

Exercise 6.2.2

Repeat the previous exercise, but this time using the `Orbit` command. Compare the timings.

6.3 Faster orbit computation

The problem with the naive approach to computing orbits is that it requires iterating over all elements of the group. This causes at least two problems:

1. The amount of work performed by `orbitNaive` is proportional to the size of the group, not the size of the orbit. Yet the orbit can be small while the group is huge. For example, for $G := \Sigma_n$ acting on $X = \{1, \ldots, n\}$, there is a single orbit of size n, but the group contains $n!$ elements. This is rather inefficient in general.
2. How do we get all elements of the group in the first place?

To improve things, we proceed as follows: Suppose $G := \langle g_1, \ldots, g_n \rangle$ is a finite group generated by the elements $g_1, \ldots, g_n$. Then every element of G can be written as a product $g_{i_1} \cdots g_{i_k}$ for suitable elements $i_1, \ldots, i_k \in \{1, \ldots, n\}$. If we now consider the orbit of some point $a \in X$, we have:

$$b \in a.G \iff \exists g \in G : b = a.g \iff \exists i_1, \ldots, i_k \in \{1, \ldots, n\} : b = a.(g_{i_1} \cdots g_{i_k}) = ((a.g_{i_1}) \ldots g_{i_{k-1}}).g_{i_k} \ .$$

This observation leads to the following algorithm:

```
1  orbit := function(G, a, act)
2      local gens, orb, x, g, b;
3      gens := GeneratorsOfGroup(G);
4      orb := [ a ];
5      for x in orb do
6          for g in gens do
7              b := act(x, g);
8              if not b in orb then
9                  Add(orb, b);
10             fi;
11         od;
12     od;
13     return orb;
14 end;
```

Exercise 6.3.1

Try to understand why this function really computes the orbit $a.G$.
Also, repeat the timing experiment from the previous two exercises.

For `orbitNaive`, we also had the problem that it was not clear how to obtain all the group elements in the first place. We actually can now apply the `orbit` function to find the group elements.

Exercise 6.3.2

Let $G := \langle g_1, \ldots, g_n \rangle$ be a group. G acts on itself by right multiplication, i.e. via $\omega(x, g) := xg$. Thus $1_G.G = G$. Use this and the orbit algorithm to write a function `elements(G)` which returns a list of all elements in G.

6.4 Even faster orbit computation

Our revise algorithm works fairly well for short orbits. But what about large orbits? To study this, we first need a way to construct actions with large orbit.

Let $n, k \in \mathbb{N}$ with $n \geq k$, let $G \leq \Sigma_n$ and $X := \{1, \ldots, n\}$. Let $X^{[k]} := \{A \subseteq X \mid |A| = k\}$ be the set of subsets of X of size k. Then G acts on $X^{[k]}$ via

$$\omega : X^{[k]} \times G \to X^{[k]} \ , \ (\{a_1, \ldots, a_k\}, g) \mapsto \{a_1^g, \ldots, a_k^g\} \ .$$

In GAP, this action is implemented by `OnSets`.

Exercise 6.4.1

Let $n, k \in \mathbb{N}$ with $n \geq k$, let $G := \Sigma_n$ and let $A_k := \{1, \ldots, k\}$.

1. Determine a formula for the size of the orbit $A_k.G$. Prove that your formula is correct.
2. For $n \in \{8, 12, 16, 18\}$ and $k := n/2$, use `orbit` and `Orbit` to compute the orbits $A_k.G$. Measure how long this takes in each case.

If everything is as it should be, you will have determined in the previous exercise that `orbit` is much, much slower than `Orbit` for large n.

Where could the problem be? Looking at the code of `orbit`, we see that the brunt of the work consists of just three computations, repeated again and again by the two loops. Specifically, the number of repetitions equals the size of the orbit times the number of generators of the group, and the operations we repeat are:

1. compute $b := a.g$;
2. check if b is already in the (partially) computed orbit;
3. if not, add b to the partial orbit.

Any implementation, no matter how clever, will have to do the first step. It turns out that adding a new element to list is very fast in GAP, and so the bottleneck must be in the innocent looking test `b in orb`.

Indeed, how can one test whether an element is in a list? Unless one is given extra information about the structure of the list, there is only one way: Go through all elements of the list, and compare them, one by one, to our new element. If the list has already size m, then on average we will need to look at $m/2$ elements before finding a match.

With some more carefully analysis, one can show that the total number of element comparisons one ends up doing when using `orbit` is in the order of m^2k, where m is the size of the orbit, and k the number of generators. So the time taken for our computation should be roughly proportional to m^2.

Exercise 6.4.2

Look at the formula and the timings you computed in the previous exercise. For each value n, compute the quotient of the time by the orbit size.

To overcome this, we need to use a better way to arrange our data, i.e., a better data structures. One which makes it more efficient to test whether some element b is already contained in the partial orbit. Adding new elements to this data structure should also be fast. In GAP, so-called dictionaries provide such a data structure. The following function uses a dictionary to replace the problematic `b in orb` test. (For more information on dictionaries in GAP, please consult the GAP manual.)

```
orbitFast := function(G, a, act)
    local gens, orb, orbDict, x, g, b;
    gens := GeneratorsOfGroup(G);
    orb := [ a ];
    orbDict := NewDictionary(a, false);
    AddDictionary(orbDict, a);
    for x in orb do
        for g in gens do
            b := act(x, g);
            if not KnowsDictionary(orbDict, b) then
                Add(orb, b);
                AddDictionary(orbDict, b);
            fi;
        od;
    od;
    return orb;
end;
```

Exercise 6.4.3

Repeat the previous exercise with `orbitFast`. How does it compare to `orbit` and `Orbit`?

Note that we now keep the partial orbit twice: once in the list `orb`, once in the dictionary `orbDict`. This causes us to waste some memory (but we normally won't need to worry about that). We need to retain the list `orb`, though, because it is important for us to be able to iterate over the elements partial orbit in precisely the order we added them; this is something lists are perfect for, but dictionaries cannot do at all: in order to be so fast at element "lookup", they have to sacrifice sequentiality of the stored data.

As a rule of thumb, there is no "perfect" data structure. Sometimes lists are best, sometimes dictionaries, sometimes some other – sometimes you need to combine multiple, as was the case here.

6.5 A bunch of useful functions for groups

In the following we present a random list of GAP functions useful when working with groups. There are many more. Unfortunately, in this course, we do not really have enough time to talk how all of these are implemented, but at least you can use them as a black-box for now.

GAP "knows" how to compute Sylow subgroups as follows:

```
gap> G := SymmetricGroup(8);;
gap> P := SylowSubgroup(G,2);
Group([ (1,2), (3,4), (1,3)(2,4), (5,6), (7,8), (5,7)(6,8), (1,5)(2,6)(3,7)(4,8) ])
gap> Size(P);           # size of P is a power of 2
128
gap> Size(G) / Size(P); # ... and |G|/|P| is not divisible by p
315
```

If you want to know how many Sylow subgroups there are, you can exploit that by Sylow's theorem, they are all conjugate together. So, we just need to ask GAP for the size of the conjugacy class.

```
gap> ccP := ConjugacyClassSubgroups(G, P);
Group( [ (1,2), (3,4), (1,3)(2,4), (5,6), (7,8), (5,7)(6,8), (1,5)(2,6)(3,7)(4,8) ] )^G
gap> Size(ccP);
315
gap> Q := SylowSubgroup(G,3);
Group([ (1,2,3), (4,5,6) ])
gap> Size(ConjugacyClassSubgroups(G, Q));
280
gap> Size(G) / Size(Q);
4480
```

You can compute normalizers, centralizers, and more.

```
gap> NQ := Normalizer(G, Q);
Group([ (1,5,3,4,2,6), (2,3), (5,6), (4,6,5), (1,3,2), (7,8) ])
gap> Size(G) = Size(NQ) * Size(ConjugacyClassSubgroups(G, Q)); # verify orbit lemma
true
gap> Size(NQ); Size(Q);
144
9
gap> Size(Centralizer(G,Q));
18
```

We talked about orbits before; of course GAP can also compute stabilizers.

```
gap> Stabilizer(G, 8, OnPoints);
Group([ (1,7), (2,7), (3,7), (4,7), (5,7), (6,7) ])
gap> last = SymmetricGroup(7);
true
gap> H := Stabilizer(G, [5,6,7,8], OnSets);
Group([ (7,8), (6,8), (5,8), (2,3), (2,4), (1,2,4) ])
gap> Size(H);
576
```

We can also try to find out more about a group, e.g. via the `StructureDescription` command.

```
gap> StructureDescription(H);
"S4 x S4"
```

Be aware that non-isomorphic groups may still have identical descriptions. But in this case, the description is unambiguous: The group H is isomorphic to the direct product $\Sigma_4 \times \Sigma_4$

Exercise 6.5.1
Try to give a theoretical explanation of why we must have $H \cong \Sigma_4 \times \Sigma_4$, based on how we defined H above.

Of course we can form direct products of groups explicitly, too; the result, however, is not necessarily what you might expect from the naive definition of direct product; in particular, the direct products often are not realized by forming tuples. This has efficiency reasons. Instead, one works with embedding and projection maps. That is: Given G_1, G_2, there are homomorphisms

$$\iota_1 : G_1 \to G_1 \times G_2 \ , \quad \iota_2 : G_2 \to G_1 \times G_2 \ , \quad \pi_1 : G_1 \times G_2 \to G_1 \ , \quad \pi_2 : G_1 \times G_2 \to G_2 \ ,$$

and these satisfy

$$\pi_1 \circ \iota_1 = \mathrm{id}_{G_1} \ , \quad \pi_2 \circ \iota_1 = \mathrm{const}_{1_{G_2}} \ , \quad \pi_1 \circ \iota_2 = \mathrm{const}_{1_{G_1}} \ , \quad \pi_2 \circ \iota_2 = \mathrm{id}_{G_2} \ .$$

```
gap> K := DirectProduct(SymmetricGroup(3), AlternatingGroup(4));
Group([ (1,2,3), (1,2), (4,5,6), (5,6,7) ])
gap> iota1 := Embedding(K,2);
1st embedding into Group([ (1,2,3), (1,2), (4,5,6), (5,6,7) ])
gap> (1,2,3)^iota1;
(4,5,6)
gap> pi1 := Projection(K,1);
1st projection of Group([ (1,2,3), (1,2), (4,5,6), (5,6,7) ])
gap> (4,5,6)^pi1;
()
gap> (4,5,6)^Projection(K,2);
(1,2,3)
```

6.6 Homework

(s6-h1) Suppose G is a group acting on a finite set X, and suppose for $a \in X$ we have $a.G = \{a_1, a_2, \ldots, a_k\}$ with $|a.G| = k \in \mathbb{N}$ and $a_1 = a$. Then a transversal for the orbit $a.G$ is a set of group elements $\{g_1, \ldots, g_k\} \subseteq G$ such that $a.g_i = a_i$ for $1 \leq i \leq k$.

Implement a function `orbTrans(G,a,act)` by which computes an orbit and a corresponding transversal. That is, it should return a list `l` such that `l[1]` is the orbit of a under G with respect to the action `act`, and `l[2]` is a transversal for that orbit. So, for $1 \leq i \leq |l[1]|$, one should have `l[1][i] = act(a, l[2][i])`.

Start from `orbitFast` and modify it to compute the transversal.

Hint: Build the transversal at the same time you create the orbit. For $x = a$, we know a transversal element, namely 1_G, as $a = a.1_G$. When looking at an arbitrary $x \in orb$, if you know $t \in G$ with $a.t = x$, then you can deduce that $x.g = (a.t).g = a.(tg)$.

(s6-h2) Let G be a group, $g \in G$ and n a positive integer. Computing g^n in the naive, straight-forward fashion requires $n-1$ multiplications. For large values of n, this can be very expensive, e.g. if g is large matrix. One can do much better. For example, to compute g^8, it suffices to compute g^2, then $g^4 = (g^2)^2$ and finally $g^8 = (g^4)^2$. So we needed three multiplications instead of seven.

This can be generalized as follows: First, represent n in binary, i.e.

$$n = b_0 + 2b_1 + 4b_2 + \cdots + 2^k b_k = \sum_{i=0}^{k} 2^i b_i$$

where $k \in \mathbb{N}$ such that $2^k \leq n < 2^{k+1}$ and $b_0, \ldots, b_k \in \{0, 1\}$. Then

$$g^n = g^{\sum_{i=0}^{k} 2^i b_i} = \prod_{i=0}^{k} g^{2^i b_i} \ .$$

Thus we need to compute the $k+1$ powers g^{2^i} and up to k product of some of these powers (depending on whether b_i is 0 or 1.

Implement a function `Pow(g,n)` which takes a group element g and an integer n, and returns g^n, computed using the scheme described above. In particular, it should use not more than approximately $2k+1$ multiplications of group elements.

Hint: Do not try to actually represent n as a binary number, doing that is inefficient. Instead, try to come up with a recursive description: b_0 is 0 if and only if n is even. Next, b_1 is 0 if and only if $(n - b_0)/2$ is even, and so on. If you arrange things in a clever fashion, you do not need to store the b_i.

Hint: Start by working out with pen and paper what needs to be done for some small values of n, say $n \in \{-1, 0, 1, 2, 19\}$.

*** next time, also ask them to do a PowMod ***

(s6-h3) The *Small Groups Library* shipped with GAP contains all groups of order at most 2000, up to isomorphism and excluding the 49 487 365 422 groups of order 1024. With `NumberSmallGroups(n)` you can find out how many isomorphism types of groups of order n there are. With `AllSmallGroups(n)` you get a list of representatives of the groups of order n; and with `AllSmallGroups(n, IsAbelian)` you get just the abelian ones. For example, to find out how many groups of order 32 are abelian, we can do this:

```
gap> Length(AllSmallGroups(32, IsAbelian));
7
```

Write a function `numAbel(n)` which computes the number of abelian groups (up to isomorphism) of order n *without* using the Small Groups library. Instead, use the classification of finite abelian groups, which states that every finite abelian group is isomorphic to a direct product of finite cyclic groups of prime power order.

Hint: First determine how many isomorphism types of abelian groups of order p^n there are, for p a prime and $n \in \mathbb{N}$. Show that this only depends on the number n, and not on the prime p.

Next, suppose A is a finite abelian group of order $p_1^{n_1} \cdots p_k^{n_k}$ for pairwise distinct primes $p_1, \ldots, p_k$. Then $A \cong A_{p_1} \times \ldots \times A_{p_k}$, where the A_{p_i} are the p_i-Sylow subgroups of A. How big are the A_{p_i}? How many possible isomorphism types for A are there?

Some GAP functions that may be very helpful for this exercise are `Factors` and `NrPartitions`.