GAC2010 — Groups, Actions and Computations

GAP session 7

Programming in GAP

1. Download Program 1 (filename "`orbitalg1.g`") from
   `http://tinyurl.com/33oxgkk`
   read it into GAP and enumerate a few orbits with it like for example in
   ```
   MyOrbit(MathieuGroup(24),1,OnPoints);
   MyOrbit(AlternatingGroup(5),[2,3],OnTuples);
   MyOrbit(GL(3,3),[Z(3),Z(3),Z(3)],OnRight);
   ```

2. Change Program 1 such that it computes a transversal in addition to the orbit.
   Hint 1: A transversal is a list `T` of group elements of the same length as the orbit `O` such that `O[i] = act(O[1],T[i])` holds for all `i`.
   Hint 2: Collect the transversal as you enumerate the orbit: For every new point, compute a group element to find this point.
   Hint 3: You need a new local variable, this needs to be initialised and whenever you add a new point to your orbit, you add a new element to the transversal list.

3. Change Program 2 (the one you created in the previous exercise) such that all Schreier generators are computed and returned in addition to the orbit and the transversal.
   Hint 1: Whenever `act(O[i],gens[j])` is already known (say as `O[k]`), then the corresponding Schreier generator is `T[i]*gens[j]*T[k]^-1`.
   Hint 2: Maintain another list S storing the Schreier generators you collect.

4. **(If you are short on time, then skip this exercise and better move on to the next one.**
   Most of the time in our programs so far is spent when looking up alleged new point in the list containing the points stored so far. This is because a linear search has to be conducted, since the list `O` is not sorted.
   The first idea to speed things up is to store the points found additionally in a sorted list. Then a binary search, which is much faster, can provide the lookup.
   Implement this idea to get a faster version of Program 3 from the previous exercise.
   Hint 1: Maintain two additional lists, one which holds the points found so far but sorted, and another which stores for each of the points in the sorted list, at which position in the original orbit list `O` it is kept. (The latter is needed to get the transversal right.)
   Hint 2: Use → `?PositionSorted` for the faster lookup in a sorted list. Note that if a point is not in the sorted list, then `PositionSorted` does not return `fail` but rather the position where the new point would have to be inserted. Do not forget that this position might be one behind the end of the list!

5. Download Program 5 (file name `partitions1.g`) and run it for a few pairs (n, k) of positive integers.
   Hint 1: A partition of n is a non-decreasing list of positive integers whose sum is n.

6. Using the ideas of Program 5, write a function that computes all partitions of n whose largest part is exactly equal to k.
   Hint 1: A similar recursive program does the job. Do the case distinction according to the second largest part.

7. Change Program 5 such that it takes an additional argument j and that it only returns the partitions of n whose largest part is at most k and which have at most j parts.
   Hint 1: Return the empty list of partitions if j is negative and decrease the argument j with every recursive call.

```
# This function computes the orbit of a point:
MyOrbit := function(g,pt,act)
  # g a group, pt a point, act an action function
  local gens,i,O,p,pos,x;
  O := [pt]; i := 1;
  gens := GeneratorsOfGroup(g);
  while i <= Length(O) do
    for x in gens do
      p := act(O[i],x);
      pos := Position(O,p);
      if pos = fail then
        Add(O,p);
      fi;
    od;
    i := i + 1;
  od;
  return rec( orbit := O );
end;
```

`orbitalg1.g`: Orbit algorithm

```
# This function computes the set of all partitions
# of n whose largest part is at most k:
MyPartitions := function(n,k)
  local l,largest,p;
  if n = 0 then return [[]]; fi;
  l := [];
  for largest in [1..Minimum(k,n)] do
      for p in MyPartitions(n-largest,largest) do
          Add(p,largest);
          Add(l,p);
      od;
  od;
  return l;
end;
```

`partitions1.g`: Computing partitions of n with largest part at most k

```
# This function computes the orbit of a point:
MyOrbit := function(g,pt,act)
  # g a group, pt a point, act an action function
  local gens,i,O,p,pos,T,x;
  O := [pt]; i := 1;
  T := [One(g)];
  gens := GeneratorsOfGroup(g);
  while i <= Length(O) do
    for x in gens do
      p := act(O[i],x);
      pos := Position(O,p);
      if pos = fail then
        Add(O,p);
        Add(T,T[i]*x);
      fi;
    od;
    i := i + 1;
  od;
  return rec( orbit := O, trans := T );
end;
```

`orbitalg2.g`: Orbit algorithm with transversal

```
# This function computes the orbit of a point:
MyOrbit := function(g,pt,act)
  # g a group, pt a point, act an action function
  local gens,i,O,p,pos,S,T,x;
  O := [pt]; i := 1;
  T := [One(g)];
  S := [];
  gens := GeneratorsOfGroup(g);
  while i <= Length(O) do
    for x in gens do
      p := act(O[i],x);
      pos := Position(O,p);
      if pos = fail then
        Add(O,p);
        Add(T,T[i]*x);
      else
        Add(S,T[i]*x*T[pos]^-1);
      fi;
    od;
    i := i + 1;
  od;
  return rec( orbit := O, trans := T, stab := S );
end;
```

`orbitalg3.g`: Orbit algorithm with Schreier generators

```
# This function computes the orbit of a point:
MyOrbit := function(g,pt,act)
  # g a group, pt a point, act an action function
  local backref,gens,i,O,p,pos,set,S,T,x;
  O := [pt]; i := 1;
  set := [pt];
  backref := [1];
  T := [One(g)];
  S := [];
  gens := GeneratorsOfGroup(g);
  while i <= Length(O) do
    for x in gens do
      p := act(O[i],x);
      pos := PositionSorted(set,p);
      if pos > Length(set) or set[pos] <> p then
        Add(O,p);
        Add(set,p,pos);
        Add(backref,Length(O),pos);
        Add(T,T[i]*x);
      else
        Add(S,T[i]*x*T[backref[pos]]^-1);
      fi;
    od;
    i := i + 1;
  od;
  return rec( orbit := O, orbitset := set,
              trans := T, stab := S );
end;
```

`orbitalg4.g`: Orbit algorithm with binary search

```
# This function computes the set of all partitions
# of n whose largest part is exactly k:
MyPartitions := function(n,k)
  local l,secondlargest,p;
  if k > n then return []; fi;
  if k = n then return [[n]]; fi;
  l := [];
  for secondlargest in [1..Minimum(k,n-k)] do
      for p in MyPartitions(n-k,secondlargest) do
          Add(p,k);
          Add(l,p);
      od;
  od;
  return l;
end;
```

`partitions2.g`: Computing partitions with largest part k

```
# This function computes the set of all partitions
# of n whose largest part is at most k with at
# most j parts:
MyPartitions := function(n,k,j)
  local l,largest,p;
  if j < 0 then return []; fi;
  if n = 0 then return [[]]; fi;
  l := [];
  for largest in [1..Minimum(k,n)] do
    for p in MyPartitions(n-largest,largest,j-1) do
      Add(p,largest);
      Add(l,p);
    od;
  od;
  return l;
end;
```

partitions3.g: Computing partitions with largest part at most k