Character Tables of Weyl Groups in GAP

Götz Pfeiffer

Lehrstuhl D für Mathematik, RWTH Aachen
Templergraben 64, D–5100 Aachen, Germany

The character tables of symmetric groups were already known to Frobenius. Meanwhile many people have contributed to the representation theory of symmetric groups and related topics. A self contained overview of the theory is given in the book *Representation Theory of the Symmetric Group* [4] by James and Kerber. We will use this book as a guideline for an implementation of the character tables of the series of Weyl groups of type A, B, and D and some related groups. We will also prove two theorems about character values of wreath products with symmetric groups (see 4.4) and Weyl groups of type D (see 5.1). For the exceptional Weyl groups of type G_2, F_4, E_6, E_7 or E_8 we will identify the characters as they are stored in the GAP library with their labels in Carter's book *Finite Groups of Lie Type: Conjugacy Classes and Complex Characters* [2].

This text is also available in a form which contains the definitions of all functions described below and which can be used as input file for GAP.

1 Case A: The Symmetric Group.

The Weyl group of type A_{n-1} is isomorphic to the symmetric group S_n on n points. Write an element $\pi \in S_n$ as a product of disjoint cycles in the natural permutation representation of S_n on n points. Then the **cycle structure** of π is described by a partition of n which consists of the lengths of the cycles of π ordered decreasingly. (Here fixed points are considered as cycles of length 1.) Recall that a **partition** of a positive integer n is a descending series of positive integers, whose sum is n. We will write a partition enclosed in brackets. For example, $[4, 2, 1]$ is a partition of 7. Partitions in GAP can be represented by lists. The empty partition will be denoted by $[\,]$.

It is easy to check the following

Lemma 1.1 ([4], (1.2.6)) *Two elements of S_n are conjugate if and only if they have the same cycle structure.*

Note that on the other hand for every partition of n there is an element of S_n with that cycle structure. Thus the partitions of n parametrize the conjugacy classes of S_n. Since we are dealing with character tables we are more interested in conjugacy classes than in single elements. By abuse of notation we will refer to both an element of S_n and its conjugacy class via π. The exact meaning will always be clear from the context.

The combinatorial basis for the construction of the character table of symmetric groups will be a mechanism to produce all partitions of a positive integer n. There is a function `Partitions` in the GAP library that might be used to compute these labels. But let us write our own function `partitions1` to see what kind of problems can arise and how they can be solved. This strategy will be very useful when in a later section we will find an efficient way to construct the character table of a symmetric group.

In order to obtain a recursive description of sets of partitions we will introduce as a second parameter a limit on the size of the parts of a partition.

Let B_k^m denote the set of all partitions of k with maximal part less than or equal to m for nonnegative k, m. Moreover let B_0^m for any m be the set which consists of only the empty partition [ ]. Then B_k^m is obtained as the union of the sets $[i] \cdot B_{k-i}^i$ for $1 \leq i \leq \min(k, m)$ where $[i] \cdot B_{k-i}^i$ denotes the set of all partitions from B_{k-i}^i preceded by an additional i-cycle. The set P_n of all partitions of n then equals B_n^n.

We will write a function `pm` which computes a set B_k^m according to the above observations by several calls to `pm(k-i, i)`. We make `pm` local to a function `partitions1` which then simply calls `pm(n, n)` in order to return all partitions of n.

```
partitions1:= function(n)

   local pm;

   pm:= function(k, m)
      local i, t, res;

      # trivial case.
      if k = 0 then
         return [[]];
      fi;

      # form the union.
      res:= [];
      for i in [1..Minimum(k, m)] do
         for t in pm(k-i, i) do
            Add(res, Concatenation([i], t));
         od;
      od;
      return res;
   end;

   # call subroutine.
   return pm(n,n);

end;
```

This function `partitions` does exactly what we wanted, but for bigger values of n it seems to be very time consuming. Note that in GAP the value of the variable `time` gives in milliseconds the time spent by the last command. The times given here are obtained on a DEC station 5000/120.

```
gap> partitions1(6);
[ [ 1, 1, 1, 1, 1, 1 ], [ 2, 1, 1, 1, 1 ], [ 2, 2, 1, 1 ],
  [ 2, 2, 2 ], [ 3, 1, 1, 1 ], [ 3, 2, 1 ], [ 3, 3 ], [ 4, 1, 1 ],
  [ 4, 2 ], [ 5, 1 ], [ 6 ] ]
gap> partitions1(20);; time;
2008
```

The problem to construct all partitions of a given n is solved by the above recursion. But to make an efficient algorithm we have to look closer at what is going on. It seems that the local function `pm` may be called several times with the same arguments. This means that especially for bigger values of n some work is done twice or even more often. To avoid such behavior we might wish to store all values ever computed by `pm` in an array or something similar. Moreover it would be desirable to get rid of the many function calls caused by the recursion. We will solve both problems in a rather pretty way. Instead of a local function we will use a list `pm` to store exactly those sets of partitions which are needed. These sets are iteratively constructed as follows.

The set B_k^m of all partitions of k with maximal part less than or equal to m can also be obtained as the union of the set B_k^{m-1} of all partitions of k with maximal part $\leq m-1$ and the set $[m] \cdot B_{k-m}^m$ of all partitions of $k-m$ with maximal part $\leq m$ preceded by an additional part m.

Obviously for $m \geq k$ every set B_k^m equals the set P_k of all partitions of k. Hence the set P_n of all partitions of n can be constructed from the sets $B_{n-k}^{\min(k,n-k)}$ for $0 \leq k \leq n$ by adding a part k respectively.

Since $\min(k, n-k) \leq n/2$ we will have to construct the necessary sets B_k^m for $0 \leq m \leq n/2$. Starting with the (empty) sets B_k^0 (except that $B_0^0 = \{[\ ]\}$) we construct the sets B_k^m for $m \leq k \leq n-m$ by using the above observations. This yields the following recursion free function `partitions`.

```
partitions:= function(n)

   local m, k, pm, t, s, res;

   if n = 0 then
      return [[]];
   fi;

   pm:= List([1..n-1], x->[]);
   for m in [1..QuoInt(n,2)] do

      # add the m-cycle.
      Add(pm[m], [m]);
      for k in [m+1..n-m] do
         for t in pm[k-m] do
            s:= [m];
            Append(s, t);
            Add(pm[k], s);
         od;
      od;
   od;

   # collect.
   res:= [];
   for k in [1..n-1] do
      for t in pm[n-k] do
         s:= [k];
```

```
            Append(s, t);
            Add(res, s);
         od;
      od;
      Add(res, [n]);

      return res;

   end;
```

Here we have also replaced the call of the function `Concatenation` by the more low level construction `s:= [m]; Append(s, t);`. The function `partitions` computes the same results as `partitions1` but in a more acceptable time for bigger values of n and the new function even isn't much more complicated than the old one. This should serve as a lesson to avoid recursions in time critical functions whenever possible.

```
   gap> partitions(6) = partitions1(6);
   true
   gap> partitions(20);; time;
   113
   gap> Partitions(20);; time;
   238
```

Note that the same algorithm can be used to compute the number of partitions of a given n. Substituting the construction of sets of partitions in the function `partitions` by a counting mechanism we get a function `nrpartitions` which shows the control structure of the general algorithm in the most compact way.

```
   nrpartitions:= function(n)

      local m, k, res;

      res:= List([1..n-1], x->0);
      for m in [1..QuoInt(n, 2)] do
         res[m]:= res[m]+1;
         for k in [m+1..n-m] do
            res[k]:= res[k] + res[k-m];
         od;
      od;

      return Sum(res)+1;

   end;
```

Now let us construct the character table of S_n as a generic table. A **generic character table** in GAP is a record similar to an ordinary character table. It does, however, not contain the actual values of characters, powermaps, etc., but functions that define these values in dependency of special parameters. The function `CharTableSpecialized` takes a generic table and a special parameter in order to construct an ordinary character table. After defining the generic table of S_n in a record `sym`, for instance, the command

```
   gap> CharTableSpecialized(sym, 8);
```

will return the ordinary character table of S_8. For a proper work a generic table should have the fields which we will install and explain now.

We begin by initializing the record `sym` with the record fields `name` and `order`. The name of the table shall be `"Symmetric"` and the order of the corresponding group shall be computed by the function `Factorial` since the order of S_n is $n!$.

```
sym:= rec(name:= "Symmetric", order:= Factorial);
```

The name for a special table i.e. for the table of S_n for a given n shall be Sn. This is managed by a function in the record field `specializedname` which assigns to each n the corresponding string.

```
sym.specializedname:= (n-> ConcatenationString("S", String(n)));
```

In a record field `text` there may as well be some explaining text to identify the table.

```
sym.text:= "generic character table for symmetric groups";
```

Moreover we tell the record `sym` that it is a generic character table and install a function `domain` which describes the valid parameters of the generic table.

```
sym.isGenericTable:= true;
sym.domain:= (n-> IsInt(n) and n > 0);
```

All remaining record fields will contain lists whose entries then will be the functions for the different parameter ranges. Since there is only one parameter range in the case of S_n these lists will all have length 1. We will prepare the bare lists now.

```
sym.classparam:= []; sym.charparam:= []; sym.centralizers:= [];
sym.orders:= []; sym.powermap:= []; sym.irreducibles:= [[]];
```

The fundamental guide through a generic table are the parametrizations of the classes and the characters which are to be stored in the record fields `classparam` and `charparam`. We already know that the classes of the symmetric group are labelled by partitions.

```
sym.classparam[1]:= partitions;
```

The function `CharTableSpecialized` will use this function in order to produce a list of parameters for the classes of a symmetric group.

The order of an element with cycle structure π for some partition π of n is just the least common multiple of the cycles or the parts of π (see [4], (1.2.14)). The evaluation of the orders in a generic table, however, expects two parameters, so we cannot just let `Lcm` be the entry in the place `orders[1]`.

```
sym.orders[1]:= function(n, lbl)
   return Lcm(lbl);
end;
```

The function `CharTableSpecialized` will call this function for every parameter `lbl` in the list of class parameters in order to produce a list of orders of representatives of the classes.

The order of the centralizer of an element $\pi \in S_n$ can best be stated in terms of numbers of cycles of equal length. Let $a_i(\pi)$ denote the number of cycles of length i of π. Then the vector $a(\pi) = (a_1(\pi), a_2(\pi), \ldots)$ is called the **type** of π. The order of the centralizer of π is computed by the following formula.

Lemma 1.2 ([4], (1.2.15)) $|C_{S_n}(\pi)| = \prod_i i^{a_i(\pi)} a_i(\pi)!$.

This number is determined (maybe not in the most efficient way) by the following function.

```
sym.centralizers[1]:= function(n, pi)

   local k, last, p, res;

   res:= 1; last:= 0; k:= 1;
   for p in pi do
      res:= res * p;
      if p = last then
         k:= k+1;
         res:= res * k;
      else
         k:= 1;
      fi;
      last:= p;
   od;

   return res;

end;
```

The powermap is also completely determined by the partitions. There is in fact a function `PowerPartition` in the GAP library that determines the partition corresponding to the k-th power of a permutation with cycle type π. This is simply done by replacing each part l of π by $d = \gcd(l, k)$ parts l/d.

```
PowerPartition:= function(pi, k)

   local res, i, d, part;

   res:= [];
   for part in pi do
      d:= GcdInt(part, k);
      for i in [1..d] do
         Add(res, part/d);
      od;
   od;
   Sort(res);

   return Reversed(res);

end;
```

We can use this function in the generic table. It has to be extended in order to return not only the power but also the number of the parameter range which is always 1 in this case.

```
sym.powermap[1]:= function(n, k, p)
   return [1, PowerPartition(k, p)];
end;
```

Having described all necessary components of the table head we now turn to the characters. For a partition $\alpha = [\alpha_1, \ldots, \alpha_m]$ of n let $S_\alpha = S_{\alpha_1} \times \cdots \times S_{\alpha_m}$ denote the corresponding **Young subgroup** of S_n. Moreover let $\tilde{\alpha}$ denote the **associated partition** of α which is obtained from α by transposing the corresponding Young diagram (see [4], (1.4.2), (1.4.3)). The following function `AssociatedPartition` computes the associated partition of a partition α.

```
AssociatedPartition := function(alpha)

   local i, j, mu;

   mu:= List([1..alpha[1]], x->0);
   for i in alpha do
      for j in [1..i] do
         mu[j]:= mu[j]+1;
      od;
   od;

   return mu;

end;
```

For a partition α let χ^α denote the common constituent of $1_{S_\alpha}^{S_n}$ and $\epsilon_{S_{\tilde\alpha}}^{S_n}$, the trivial character of S_α and the sign character of $S_{\tilde\alpha}$ induced to S_n. Then χ^α is in fact an irreducible character of S_n and

Proposition 1.3 ([4], (2.1.11)) *The characters χ^α run through a complete system of pairwise different and irreducible characters of S_n, if α runs through all partitions of n.*

Hence the characters of S_n are labelled by partitions of n, too.

```
sym.charparam[1]:= partitions;
```

The most interesting part is the computation of the character values. A nice way to describe the character values of S_n in terms of partitions is given by the Murnaghan–Nakayama formula. We need some further notation to state it. At each node (i, j) of the Young diagram of a partition α of n (that is for each i, j with $\alpha_i \geq j$) there is a **hook** which consists of all nodes to the right of (i, j) and all nodes below (i, j). Its **hooklength** (the number of nodes involved in the hook) is denoted by h_{ij}^α. All nodes below (i, j) form the **leg** of the hook with leglength l_{ij}^α. Removing the hook from the diagram results in a diagram of another partition only if the rows of the diagram are sorted again. The same diagram is obtained if instead of the hook itself the corresponding **rim hook** (denoted by R_{ij}^α) is removed from the original diagram (see [4], (2.3.18–20)).

Theorem 1.4 (Murnaghan–Nakayama formula [4], (2.4.7)) *Let α be a partition of n and let $\pi \in S_n$ with $a_k(\pi) > 0$ for a fixed $k \leq n$. Let $\rho \in S_{n-k}$ be of type $a(\rho)$ where*

$$a_i(\rho) = \begin{cases} a_k(\pi) - 1 & \text{if } i = k, \\ a_i(\pi) & \text{otherwise.} \end{cases}$$

Define $\chi^{[\,]} = 1$. Then

$$\chi^\alpha(\pi) = \sum_{h_{ij}^\alpha = k} (-1)^{l_{ij}^\alpha} \chi^{\alpha - R_{ij}^\alpha}(\rho).$$

Here the sum is taken over all nodes (i, j) in the tableau of α, where the corner of a hook of length k is found.

We will now implement that formula in a straightforward way just to show how easy even such a seemingly complicated formula can be handled in GAP. Then we will improve the program step by step.

The formula tells us that we need three little functions in advance, the functions `hooklength`, `leglength` and `unrimmed` which will return the hooklength and the leglength of a hook and the partition which results from removing a rim hook.

Here is a function `leglength` which returns the leglength of a hook at the node (i, j) in a partition α.

```
leglength:= function(alpha, i, j)

   local ll, lp, k;

   k:= i+1;
   lp:= Length(alpha);
   ll:= 0;
   while  k <= lp  and  alpha[k] >= j  do
      ll:= ll + 1;
      k:= k+1;
   od;

   return ll;

end;
```

A short extension of this last function may serve as the function `hooklength` which returns the hooklength of a hook at the node (i, j) in a partition α.

```
hooklength:= function(alpha, i, j)

   local hl, lp, k;

   hl:= alpha[i] - j + 1;
   if hl <= 0 then
      return 0;
   fi;
   k:= i+1;
   lp:= Length(alpha);

   while  k <= lp  and  alpha[k] >= j  do
      hl:= hl + 1;
      k:= k + 1;
   od;

   return hl;

end;
```

The following function `unrimmed` will return the partition which results from α by removing the rim hook R^{α}_{ij}. Removing a rim hook from a Young tableau is the same as removing the hook itself and sorting the list of numbers to give again a partition.

```
unrimmed:= function(alpha, i, j)

   local k, lp, rho;

   lp:= Length(alpha);
```

```
        rho:= [];
        for k in [1..i-1] do
           rho[k]:= alpha[k];
        od;

        k:= i;

        # a special case first.
        if j = 1 then
           while  k < lp  and  alpha[k+1] >= j+1  do
              rho[k]:= alpha[k+1] - 1;
              k:= k+1;
           od;

           # don't add trailing zeros.
           return rho;

        fi;

        while  k < lp  and  alpha[k+1] >= j  do
           rho[k]:= alpha[k+1] - 1;
           k:= k+1;
        od;

        rho[k]:= j-1;

        for k in [k+1..lp] do
           rho[k]:= alpha[k];
        od;

        return rho;

     end;
```

Now we are in a position to write the function `chi1` which computes single character values. This function can be implemented almost in the same way as it is written down as a formula. The formula even tells us which local variables are needed. For the value of k we will take the longest cycle of π in order to continue the recursion in the smallest possible subgroup.

```
     chi1:= function(n, alpha, pi)

        local i, j, k, rho, val;

        # termination condition.
        if pi = [] then
           return 1;
        fi;

        # get length of longest cycle.
        k:= pi[1];

        # construct rho.
        rho:= Sublist(pi, [2..Length(pi)]);

        val:= 0;

        # loop over the Young diagram.
        for i in [1..Length(alpha)] do
           for j in [1..alpha[i]] do
              if hooklength(alpha, i, j) = k then

                 #  enter recursion.
                 val:= val + (-1)^leglength(alpha, i, j)
                      * chi1(n-k, unrimmed(alpha, i, j), rho);
              fi;
           od;
        od;

        #  return the result.
        return val;

     end;
```

Then we install the function in the record field `irreducibles` of `sym`.

```
sym.irreducibles[1][1]:= chi1;
```

Voilà. We now can use this generic table to produce character tables of symmetric groups for specialized values of n. Let's try S_4 as a first example.

```
gap> s4:= CharTableSpecialized(sym, 4);
rec( name := "S4", order := 24, centralizers := [ 24, 4, 8, 3, 4
 ], orders := [ 1, 2, 2, 3, 4 ], powermap :=
[ , [ 1, 1, 1, 4, 3 ], [ 1, 2, 3, 1, 5 ] ], irreducibles :=
[ [ 1, -1, 1, 1, -1 ], [ 3, -1, -1, 0, 1 ], [ 2, 0, 2, -1, 0 ],
  [ 3, 1, -1, 0, -1 ], [ 1, 1, 1, 1, 1 ] ], classparam :=
[ [ 1, [ 1, 1, 1, 1 ] ], [ 1, [ 2, 1, 1 ] ], [ 1, [ 2, 2 ] ],
  [ 1, [ 3, 1 ] ], [ 1, [ 4 ] ] ], irredinfo := [ rec(
      charparam := [ 1, [ 1, 1, 1, 1 ] ] ), rec(
      charparam := [ 1, [ 2, 1, 1 ] ] ), rec(
      charparam := [ 1, [ 2, 2 ] ] ), rec(
      charparam := [ 1, [ 3, 1 ] ] ), rec(
      charparam := [ 1, [ 4 ] ] )
 ], text := "computed using generic character table for symmetric g\
roups", classes := [ 1, 6, 3, 8, 6 ], operations := CharTableOps )
```

Among other checks for consistency the orthogonality relations are tested by the function `TestCharTable`. The function `DisplayCharTable` prints a formated version of a character table.

```
gap> TestCharTable(s4);
true
gap> DisplayCharTable(s4);
S4

     2  3  2  3  .  2
     3  1  .  .  1  .

       1a 2a 2b 3a 4a
    2P 1a 1a 1a 3a 2b
    3P 1a 2a 2b 1a 4a

X.1     1 -1  1  1 -1
X.2     3 -1 -1  .  1
X.3     2  .  2 -1  .
X.4     3  1 -1  . -1
X.5     1  1  1  1  1
```

Note that due to the chosen parametrization of the characters the trivial character is not the first but the last character of our table. But this doesn't matter since there is no place in GAP where it is assumed that the trivial character is the first one.

The computation of tables for larger values of n takes considerably more time. Let us try S_8 as next example.

```
gap> s8:= CharTableSpecialized(sym ,8);; time;
11843
```

Now it is time to think about possible speed up of time critical functions. The worst case of the recursion is when π consists of only cycles of length one, that is when π is the identity and the character value in question is the degree of the character. But the character degree of χ^α may be computed directly as the quotient of the order of S_n and the product of all hook lengths of α.

Proposition 1.5 ([4], (2.3.21)) *Let α be a partition of n. Then*

$$\chi^\alpha(1) = \frac{n!}{\prod_{i,j} h_{ij}^\alpha}.$$

The corresponding GAP function is immediately written down.

```
symdegree:= function(n, alpha)

   local i, j, prod;

   prod:= 1;
   for i in [1..Length(alpha)] do
      for j in [1..alpha[i]] do
         prod:= prod * hooklength(alpha, i, j);
      od;
   od;

   return  Factorial(n) / prod;

end;
```

There is another almost trivial case. If π consists of only the n-cycle there is no need for further recursion. The character value then depends only on the existence of an n–hook (there is at most one!) in α and its leg parity.

Proposition 1.6 ([4], (2.3.17)) *Let α be a partition of n. Then*

$$\chi^\alpha((1,\ldots,n)) = \begin{cases} (-1)^r & \text{if } \alpha = [n-r, 1^r], 0 \le r \le n-1 \\ 0 & \text{otherwise.} \end{cases}$$

Denote by $l(\alpha)$ the **length** of the partition α (i.e. the number of parts). If there is an n-hook then it must be all of α, that is $\alpha_1 + l(\alpha) - 1 = n$. In that case the leglength of the hook is given by $l(\alpha) - 1$.

These two new insights are now worked into a new function `chi2`. They will thereby replace the termination condition `if pi = [] ...` of `chi1`.

```
chi2:= function(n, alpha, pi)

   local i, j, k, rho, val;

   # get length of longest cycle.
   k:= pi[1];

   # degree case.
   if k = 1 then
      return symdegree(n, alpha);
   fi;

   # almost trivial case.
   if k = n then
      if hooklength(alpha, 1, 1) = k then
         return (-1)^(Length(alpha)-1);
      else
         return 0;
      fi;
   fi;

   val:= 0;
   rho:= Sublist(pi, [2..Length(pi)]);

   # loop over the Young diagram.
   for i in [1..Length(alpha)] do
      for j in [1..alpha[i]] do
         if hooklength(alpha, i, j) = k then

            # enter recursion.
            val:= val + (-1)^leglength(alpha, i, j)
                 * chi2(n-k, unrimmed(alpha, i, j), rho);
         fi;
      od;
   od;

   # return the result.
   return val;

end;
```

Assigning the new function chi2 to its place in the generic table will replace the old function chi1 in the record sym. The later function is, however, still available under its name chi1.

```
sym.irreducibles[1][1]:= chi2;
```

If we now try S_8 again we will see that we saved two third of the time by only slight changes.

```
gap> s8:= CharTableSpecialized(sym ,8);; time;
4132
```

Four seconds is almost acceptable (it takes longer to restore the table of S_8 from the library). But S_8 is still a small table with only 22 classes and characters. And we want to be able to compute larger tables in acceptable time. So let's have a look at the profile in order to find out where the time is spent.

```
gap> Profile(true);
gap> s8:= CharTableSpecialized(sym, 8);;
gap> Profile(false);
gap> Profile();
 count    time percent time/call child function
  1509    1507      21        0   4734  chi2
  7192    1432      20        0   1739  hooklength
 15707     575       8        0    420  Length
   344     413       5        1   1404  Gcd
  1025     389       5        0    446  unrimmed
   356     296       4        0    817  symdegree
  3408     274       3        0   5145  Add
   712     184       2        0    393  ForAll
  1025     165       2        0    213  leglength
   899     157       2        0    288  Sublist
   367     148       2        0    375  DefaultRing
    88     145       2        1   1743  PowerPartition
   ...
```

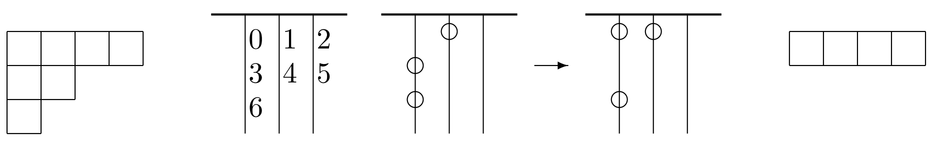
Almost half of the time is spent in the functions `chi2` (21 percent of the total time) and `hooklength` (20 percent). Moreover we see that `hooklength` is called 7192 times for only 1509 calls of `chi2`. (This was even worse with the old `chi1` where `hooklength` was called 18486 times for 7826 calls of `chi1`, so our first changes met the right place.) If we now look back at the code of `chi2` we will see where this comes from.

```
# loop over the Young diagram
for i in [1..Length(alpha)] do
   for j in [1..alpha[i]] do
      if hooklength(alpha, i, j) = k then
         ...
```

These nested `for` loops around the call of `hooklength` are a mathematically exact picture of the above Murnaghan–Nakayama formula. But what they will cause the computer to do is obviously too much. There is no need to run trough the whole Young diagram for α every time and ask for the hooklength. There is, for example, at most one hook of length k on each row of a diagram.

The functions `leglength` and `unrimmed` also seem very awkward. Ask an expert and you will find out that a more comfortable way to deal with hooks and leglengths is the concept of beta sets (see [4], (2.7)).

A **beta set** is a set of numbers which arises from a partition by reversing the order and adding a vector $[0, 1, 2, \ldots]$ of the same length. Since the reversed partition is allowed to have leading zeros, its beta set is not uniquely determined. Each beta set, however, determines a unique partition. For example a beta set for the partition $[4, 2, 1]$ is given by $[1, 3, 6]$, another one by $[0, 1, 3, 5, 8]$. In order to remove a k–hook from the corresponding Young diagram the beta-numbers are placed as beads on k strings.



To find a removable k-hook now simply means to find a free place for a bead one step up on its string, the hook is then removed by lifting this bead. (You see how this process can produce leading zeros.)

For the implementation of the Murnaghan–Nakayama formula this means a substantial simplification. We no longer have to loop over the two dimensional Young diagram but only over the corresponding beta set and find beads which can be moved k positions downward. We even do no longer need a function `hooklength`. Furthermore observe that the leglength of a hook equals exactly the number of passed beads by a down moving bead.

So we will now change the concept and represent the partitions referring to the characters by beta sets. In GAP we will represent a beta set by a set. The first function we will need is the function `BetaSet` which converts a partition into a beta set.

```
BetaSet:= function(alpha)

   local i, la, beta;

   la:= Length(alpha);
   beta:= [];
   for i in [0 .. la-1] do
      Add(beta, alpha[la-i] + i);
   od;

   return beta;

end;
```

There should also be a function which converts a beta set into a partition but we will not need such a function here.

Next we have to write a new version of `symdegree` which deals with beta sets instead of partitions. For every bead in a beta set all free positions have to be found to which this bead can be moved down. The first free place is determined by the smallest number greater or equal to zero which is not contained in the beta set. The function `symdegreebeta` will first find this offset `o` which is zero for all beta sets constructed by `BetaSet`.

```
symdegreebeta:= function(n, beta)

   local i, j, o, prod;

   prod:= 1;

   # determine offset.
   o:= 0;
   while beta[o+1] = o do
      o:= o+1;
   od;

   # find all beads.
   for i in [o+1..Length(beta)] do
      prod:= prod * (beta[i] - o);

      # find other free places.
      for j in [o+1..beta[i]-1] do
```

```
            if  not j in beta then
               prod:= prod * (beta[i]-j);
            fi;
         od;

      od;

      return Factorial(n)/prod;

   end;
```

Since it is so easy to remove a rim hook from a beta set by just moving a bead there will be no special function for that purpose in the new function `chi3`. In this version we will also use the fact that not really the leglength of a hook is needed but only its parity. We will not write a special function for this computation.

```
   chi3:= function(n, beta, pi)

      local i, j, k, o, gamma, rho, val, sign;

      # get length of longest cycle.
      k:= pi[1];

      # degree case.
      if  k = 1  then
         return symdegreebeta(n, beta);
      fi;

      # determine offset.
      o:= 0;
      while beta[o+1] = o do
         o:= o+1;
      od;

      # almost trivial case.
      if k = n then
         if  n + o in beta then
            return (-1)^(Size(beta)+o+1);
         else
            return 0;
         fi;
      fi;

      rho:= Sublist(pi, [2..Length(pi)]);
      val:= 0;

      # loop over the beta set.
      for i in beta do
         if  i >= k+o and not i-k in beta  then

            # compute the leg parity.
            sign:= 1;
            for j in [i-k+1..i-1] do
               if j in beta then
                  sign:= -sign;
               fi;
            od;

            # compute new beta set.
            gamma:= Difference(beta, [i]);
            AddSet(gamma, i-k);

            # enter recursion.
            val:= val + sign * chi3(n-k, gamma, rho);
         fi;
      od;

      # return the result.
      return val;

   end;
```

Note that it would be a fatal mistake just to move down beads in the `beta` of the function. Then different branches of the recursion would work with the same `beta` that was changed from very different places.

Since `chi3` expects a beta set as the parameter for the character but the parameters still are partitions we have to wrap it before it is installed in the record `sym`.

```
   sym.irreducibles[1][1]:= function(n, alpha, pi)
      return chi3(n, BetaSet(alpha), pi);
   end;
```

Again we check the time needed for the character table of S_8 and find out that again we saved a considerable amount.

```
   gap> s8:= CharTableSpecialized(sym, 8);; time;
   2719
```

The record `sym` we defined in this section is part of the GAP library under the name `CharTableSymmetric`. It is returned by the command `CharTable("Symmetric")` and, for instance, the command `CharTable("Symmetric", 5)` will return the character table of S_5 by applying `CharTableSpecialized` to this record.

The functions which are installed there can be used by other functions. This is done, for example, by `CharTableSpecialized`. But it is also possible to construct only parts of a character table with these functions. We will do this in the next section where the generic character table of alternating groups is obtained by restricting certain characters from symmetric groups. After that we will return to symmetric groups once more.

2 Alternating Groups.

The alternating group A_n on n points is a normal subgroup of index two of the symmetric group S_n. So it is very easy to derive its character table from that of the symmetric group by Clifford theory. This is a quick but unusual approach since Clifford theory normally is used to obtain characters of the bigger group from those of the normal subgroup.

We begin by initializing a record `alt` with a name and a function for the order of an alternating group.

```
   alt:= rec(name:= "Alternating");
   alt.order:= function(n)
      return Factorial(n)/2;
```

```
    end;
```

The specialized name of the table of A_n shall be An. The fields `text`, `domain`, and `isGenericTable` are filled in a similar way as in the preceding section.

```
    alt.specializedname:= (n-> ConcatenationString("A", String(n)));
    alt.text:= "generic character table for alternating groups";
    alt.isGenericTable:= true;
    alt.domain:= (n-> IsInt(n) and n > 2);
```

Moreover we will initialize the remaining lists of functions which will have length one in this case, too.

```
    alt.classparam:= []; alt.charparam:= []; alt.centralizers:= [];
    alt.orders:= []; alt.powermap:= []; alt.irreducibles:= [[]];
```

The alternating group is the kernel of the homomorphism from the symmetric group into $\{1, -1\}$ which maps each permutation to its sign. The sign of a permutation is the parity of the number of transpositions you need to write the permutation as their product. An n-cycle needs $n - 1$ transpositions so its sign is $(-1)^{n-1}$. The sign can be computed from the cycle type and therefore from the partition which labels a class. This is done by the function `SignPartition` which takes a partition as its argument and returns the sign of an element with such a cycle structure.

```
    SignPartition := function(pi)
       return (-1)^(Sum(pi) - Length(pi));
    end;
```

Next thing to investigate: Which classes do split in A_n? If a class does not split then an element of that class has exactly as many conjugates in A_n as it has in S_n. Since the order of the group equals the product of the length of the class and the order of the centralizer of that element this means on the other hand that the centralizer of such an element in A_n is only half as big as in S_n. Consequently a class of S_n splits if and only if the centralizer of one of its elements in S_n lies completely in A_n.

So we have to consider centralizers of permutations in symmetric groups. Writing an element g of S_n as a product of disjoint cycles we see that every power of every single cycle lies in the centralizer of g. If g contains an even cycle then this cycle lies in the centralizer of g but not in A_n. So let us assume that g only consists of odd cycles. If g contains two cycles of the same length then there is a product of that many transpositions interchanging these two cycles and centralizing g. Again this product of an odd number of transpositions does not lie in A_n. So assume that g is a product of cycles with pairwise different odd lengths. Then the centralizer of g is exactly the group generated by the single cycles of g which is a subgroup of A_n since all cycles are odd. This yields the following

Lemma 2.1 ([4](1.2.10)) *The conjugacy class of elements of S_n with cycle structure π splits into two A_n-classes of equal size if and only if $n > 1$ and the parts of π are pairwise different and odd.*

The following function will return the list of class parameters for A_n. Instead of the partition it will return a pair of a partition and a sign in the case of split classes. It has a local function `pdodd` which determines whether a partition consists of pairwise different, odd parts.

```
alt.classparam[1]:= function(n)

   local labels, pi, pdodd;

   pdodd:= function(pi)
      local i;
      if pi[1] mod 2 = 0 then
         return false;
      fi;
      for i in [2..Length(pi)] do
         if pi[i] = pi[i-1] or pi[i] mod 2 = 0 then
            return false;
         fi;
      od;
      return true;
   end;

   labels:= [];
   for pi in Partitions(n) do
      if SignPartition(pi) = 1 then
         if pdodd(pi) then
            Add(labels, [pi, '+']);
            Add(labels, [pi, '-']);
         else
            Add(labels, pi);
         fi;
      fi;
   od;

   return labels;

end;
```

We will also prepare a short function `issplit` which recognizes whether a label belongs to a split class.

```
issplit:= function(lbl)
   return Length(lbl) = 2 and not IsInt(lbl[2]);
end;
```

The order of an element in A_n is still determined by its cycle structure. But we have to check whether the label contains an additional sign.

```
alt.orders[1]:= function(n, lbl)
   if issplit(lbl) then
      lbl:= lbl[1];
   fi;
   return Lcm(lbl);
end;
```

In most cases the centralizer of an element in A_n is half as big as its centralizer in S_n except for the split classes.

```
alt.centralizers[1]:= function(n, lbl)
   local cen;
   if issplit(lbl) then
      return sym.centralizers[1](n, lbl[1]);
   else
      return sym.centralizers[1](n, lbl)/2;
   fi;
   end;
```

The computation of the powermap then has to consider some special cases and is slightly more complicated than in the case of S_n. On the nonsplit classes everything works like in the symmetric group. On the split classes we have to check whether the power of the element will be of a smaller order. This happens if and only if the prime divides one of the parts of the label. In that case this part will be replaced by several smaller parts, hence the resulting class is not a split class and labelled only by a partition. If the prime and the order of the element are coprime we have to check via the character values whether the two classes of this split pair are interchanged by the powermap. So we first have to investigate the character values.

The sign character σ of S_n describes the homomorphism from S_n to the multiplicative group $\{-1, 1\}$ with kernel A_n. Hence the characters χ and $\sigma\chi$ will restrict to the same character of A_n for every character χ of S_n. Tensoring with the sign character σ acts on the labelling partitions by transposing the corresponding Young diagram. The characters χ and $\sigma\chi$ will even restrict to an irreducible character of A_n unless the partition of χ equals its associated partition. In that case the restriction of χ to A_n will be the sum of two irreducible characters.

Theorem 2.2 ([4], (2.5.7)) *Let α be a partition of $n > 1$.*

1. *If α is different from its associated partition $\tilde{\alpha}$ then $\chi^{\alpha}|_{A_n} = \chi^{\tilde{\alpha}}|_{A_n}$ is irreducible.*
2. *If α is self–associated then $\chi^{\alpha}|_{A_n} = \chi^{\tilde{\alpha}}|_{A_n}$ splits into two irreducible and conjugate characters χ^{α}_{+} and χ^{α}_{-} of A_n.*

The following function will compute the labels for the characters of A_n. It will collect a list of labels from the set of all partitions of n. Self–associated partitions will be entered twice together with an additional sign. The labels for the characters are chosen in such a way that `issplit` can be applied to them, too. From every pair of associated partitions only the smaller one (according to the GAP ordering of its objects which is in this case lexicographic) will be entered in the list of labels.

```
   alt.charparam[1]:= function(n)
      local alpha, labels;

      labels:= [];
      for alpha in partitions(n) do
         if alpha = AssociatedPartition(alpha) then
            Add(labels, [alpha, '+']);
            Add(labels, [alpha, '-']);
         elif alpha < AssociatedPartition(alpha) then
            Add(labels, alpha);
         fi;
      od;
      return labels;
   end;
```

With these informations we can use the functions developed for symmetric groups to compute large parts of the character table of alternating groups. The remaining question is: What happens with the split characters on the split classes? The following nice result shows that things are not very complicated in this case.

The Young diagram of a self–associated partition α is symmetric with respect to its main diagonal. This means that the diagram consists of hooks with equal leglength and hooklength along the diagonal. Let $h(\alpha) = [h_{11}^{\alpha}, h_{22}^{\alpha}, \ldots]$ denote the corresponding **hook partition**. Note that $h(\alpha)$ consists of pairwise different odd parts, hence it describes a split conjugacy class of S_n. It turns out that only the character values of χ^{α} on this class need a special investigation while all other character values of $\chi_{\pm}^{\alpha}$ are half the value of χ^{α}.

The character value $\chi^{\alpha}(h(\alpha))$ in S_n is immediately computed by the Murnaghan–Nakayama formula (1.4).

Lemma 2.3 ([4], (2.5.12)) *If α is a self–associated partition of n and k denotes the length of the main diagonal of the Young diagram of α then*

$$\chi^{\alpha}(h(\alpha)) = (-1)^{(n-k)/2}$$

The interesting character value is then given by

Theorem 2.4 ([4], (2.5.13)) *If α is a self–associated partition of of $n > 1$ then the values of $\chi_{\pm}^{\alpha}$ are*

$$\chi_{\epsilon}^{\alpha}(h(\alpha)_{\delta}) = \frac{1}{2}\left(\chi^{\alpha}(h(\alpha)) + \epsilon\delta\sqrt{\chi^{\alpha}(h(\alpha))\prod_i h_{ii}^{\alpha}}\right),$$

for a suitable numbering of the constituents of $\chi^{\alpha}|_{A_n}$, while on all other classes of A_n with cycle structures $\gamma \neq h(\alpha)$ we have

$$\chi_{\pm}^{\alpha}(\gamma) = \chi^{\alpha}(\gamma)/2$$

and $\chi^{\alpha}(\gamma)$ is an even integer.

There is a function `ER` in the GAP library which computes square roots of integers in terms of cyclotomic numbers. But there is a more appropriate way to describe the above character value, which lies in a quadratic extension of the rationals. The ATLAS irrationalities b_N (see [3], p. xxvii) describe the most common character values from those fields.

$$b_N = \begin{cases} \frac{1}{2}(-1+\sqrt{N}) & \text{if } N \equiv 1 (\mathrm{mod} 4), \\ \frac{1}{2}(-1+i\sqrt{N}) & \text{if } N \equiv -1 (\mathrm{mod} 4). \end{cases}$$

These expressions are very close to the above character values. Note that for a suitable choice of ϵ and δ such that $\epsilon\delta\chi^{\alpha}(h(\alpha)) = -1$ we have

$$\chi_{\epsilon}^{\alpha}(h(\alpha)_{\delta}) = -\chi^{\alpha}(h(\alpha)) \cdot \frac{1}{2}\left(-1 + \sqrt{\chi^{\alpha}(h(\alpha))} \cdot \sqrt{\prod_i h_{ii}^{\alpha}}\right).$$

Denote $\prod h^{\alpha}_{ii}$ by N and observe that by (2.3) always $(-1)^{(n-k)/2} = \chi^{\alpha}(h(\alpha)) \equiv N(\text{mod} 4)$. Then we have

$$\chi^{\alpha}_{\epsilon}(h(\alpha)_{\delta}) = \begin{cases} -\chi^{\alpha}(h(\alpha)) \cdot b_N & \text{for } \epsilon = \delta \\ \chi^{\alpha}(h(\alpha)) \cdot (1 + b_N) & \text{otherwise,} \end{cases}$$

for a suitable labelling of the constituents of $\chi^{\alpha}|_{A_n}$, that might be different from that in (2.4). The values b_N are computed by the GAP function `EB`.

Now we can completely describe the powermap of alternating groups. The question whether two split classes are exchanged by a p-th powermap can be reduced to the question whether the two irrational character values on that class are p-th Galois conjugates, that is whether $-b_N^{*p} = 1 + b_N$.

```
alt.powermap[1]:= function(n, lbl, p)
   local val, prod;

   # split case.
   if issplit(lbl) then
      prod:= Product(lbl[1]);

      # coprime case needs complicated check.
      if prod mod p <> 0 then
         val:= EB(prod);
         if val+1 = -GaloisCyc(val, p) then
            if lbl[2] = '+' then
               return [1, [lbl[1], '-']];
            else
               return [1, [lbl[1], '+']];
            fi;
         else
            return [1, lbl];
         fi;
      else
         return [1, PowerPartition(lbl[1], p)];
      fi;
   fi;

   # ordinary case.
   return [1, PowerPartition(lbl, p)];

end;
```

Due to the less homogeneous parametrization of the classes and the characters of the alternating groups the function that computes the character values will have to distinguish several cases. But we will use the function `sym.irreducibles[1][1]` from the symmetric group. This will work with either function in that place. Since the value of χ^{α} is even on all uncomplicated classes for a split α we will not have to check whether π equals the hook partition of α. The classes with the special values are detected by a value of 1 or -1 for χ^{α}.

```
alt.irreducibles[1][1]:= function(n, alpha, pi)

   local val;

   if issplit(alpha) then
```

```
            if issplit(pi) then
               val:= sym.irreducibles[1][1](n, alpha[1], pi[1]);
               if val in [-1, 1] then
                  if alpha[2] = pi[2] then
                     val:= -val * EB(Product(pi[1]));
                  else
                     val:= val * (1 + EB(Product(pi[1])));
                  fi;
               else
                  val:=  val/2;
               fi;
            else
               val:= sym.irreducibles[1][1](n, alpha[1], pi)/2;
            fi;

         else
            if issplit(pi) then
               val:= sym.irreducibles[1][1](n, alpha, pi[1]);
            else
               val:= sym.irreducibles[1][1](n, alpha, pi);
            fi;
         fi;

         return val;

      end;
```

Now we can use this generic table with the function `CharTableSpecialized`.

```
    gap> a5:= CharTableSpecialized(alt, 5);
    rec( name := "A5", order := 60, centralizers := [ 60, 4, 3, 5, 5
     ], orders := [ 1, 2, 3, 5, 5 ], powermap :=
    [ , [ 1, 1, 3, 5, 4 ], [ 1, 2, 1, 5, 4 ],, [ 1, 2, 3, 1, 1 ]
     ], irreducibles := [ [ 1, 1, 1, 1, 1 ], [ 4, 0, 1, -1, -1 ],
      [ 5, 1, -1, 0, 0 ], [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ],
      [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ] ], classparam :=
    [ [ 1, [ 1, 1, 1, 1, 1 ] ], [ 1, [ 2, 2, 1 ] ], [ 1, [ 3, 1, 1 ] ],
      [ 1, [ [ 5 ], '+' ] ], [ 1, [ [ 5 ], '-' ] ] ], irredinfo :=
    [ rec(
          charparam := [ 1, [ 1, 1, 1, 1, 1 ] ] ), rec(
          charparam := [ 1, [ 2, 1, 1, 1 ] ] ), rec(
          charparam := [ 1, [ 2, 2, 1 ] ] ), rec(
          charparam := [ 1, [ [ 3, 1, 1 ], '+' ] ] ), rec(
          charparam := [ 1, [ [ 3, 1, 1 ], '-' ] ] )
     ], text := "computed using generic character table for alternating\
     groups", classes := [ 1, 15, 20, 12, 12
     ], operations := CharTableOps )
```

Even if the function `TestCharTable` returns `true` for this character table there is no guarantee that it really is a correct character table. But the displayed version can easily be compared with the known character table of A_5 and we see that at least in this case the result is correct.

```
    gap> TestCharTable(a5);
    true
    gap> DisplayCharTable(a5);
    A5

       2  2  2  .   .   .
       3  1  .  1   .   .
```

```
   5  1  .  .   1   1

      1a 2a 3a  5a  5b
   2P 1a 1a 3a  5b  5a
   3P 1a 2a 1a  5b  5a
   5P 1a 2a 3a  1a  1a

X.1    1  1  1   1   1
X.2    4  .  1  -1  -1
X.3    5  1 -1   .   .
X.4    3 -1  .   A  *A
X.5    3 -1  .  *A   A

A = -E(5)-E(5)^4
  = (1-ER(5))/2 = -b5
```

The character table of the alternating group of degree 8 is constructed in about one second.

```
gap> a8:= CharTableSpecialized(alt, 8);; time;
1152
```

The record `alt` we defined in this section is part of the GAP library where it is called `CharTableAlternating`. A call of `CharTable` with first argument `"Alternating"` will refer to that record.

This technique of restricting a known generic character table to a normal subgroup of index two will be applied again in section 5.

3 Symmetric Groups Revisited.

We have found many improvements for the way to compute single character values of symmetric groups. And the resulting function `chi3` is pretty fast. Nonetheless there can be done more if one is interested in the whole character table. There seem to be many calls to `chi3` with the same arguments during the computation of a complete character table. So much of the work is done more than once. We might now consider administrative tricks to store all values that are computed and if a value is wanted a second time rather use the stored value than do the computation again. This reminds much of the problems we had in the first section with the construction of the set of all partitions. There we found a nice way to avoid multiple computations of some values. Why should this not work with the character values.

So let us have a closer look a the original formula

$$\chi^\alpha(\pi) = \sum_{h^\alpha_{ij}=k} (-1)^{l^\alpha_{ij}} \chi^{\alpha - R^\alpha_{ij}}(\rho)$$

to see what it means for the whole matrix of character values. Let π be the partition $[4,2,1]$ of 7, for instance. If we decide to let $k=4$ regardless of α then the formula claims that every character value of that class of S_7 is computed from some character values of the class $\rho = [2,1]$ of S_3. In other words: The column $[4,2,1]$ of the character table of S_7 arises from the column $[2,1]$ of S_3 via some combinatorial mechanism encoded in the labels of the characters.

More generally every column of the character table of S_n corresponding to a partition with maximal part k arises from a column of the character table of S_{n-k} whose corresponding partition has a maximal part less than or equal to $\min(k, n-k)$. But this is only a restatement of the rules for sets of partitions that we found in the first section. There we used these rules for a recursion free construction of all partitions of n with the function `partitions`.

In order to get a recursion free construction of the character table of S_n we will have to describe the combinatorial mechanism which connects the columns of different symmetric groups. For that purpose let `scheme` be a list of length n. For every partition α of some $m \leq n$ and every $k \leq m$ this list shall contain the information, which partitions result from α by removing a k-hook and what the leg parity of that hook is. Suppose that `i` is the position of α in the list of all partitions of m. Then `scheme[m][i][k]` will be a list of numbers. The absolute value of such a number then is the position in the list of all partitions of $m-k$ of a partition that results from α by removing a k-hook, and the sign of this number will be the corresponding leg parity. If `scheme[m][i][k]` is the complete list of all such numbers then this is all the information that is needed.

The following function `InductionScheme` will construct the multi–dimensional array `scheme`. Again we prefer beta sets to find all the hooks. In order to identify a beta set in a list of beta sets in a standard form leading zeros have to be removed when they arise. The local function `hooks` will run over a single beta set β, encode every hook it encounters and add this to `scheme`.

```
InductionScheme := function(n)

   local scheme, pm, i, beta, hooks;

   pm:= [];
   scheme:= [];

   # how to encode all hooks.
   hooks:= function(beta, m)
      local i, j, l, gamma, hks, sign;

      hks:= [];
      for i in [1..m] do
         hks[i]:= [];
      od;

      for  i in  beta do

         sign:= 1;

         for j in [1..i]  do
            if i-j in beta then
               sign:= -sign;
            else
               if j = m then
                  Add(hks[m], sign);
               else
                  gamma:= Difference(beta, [i]);
                  AddSet(gamma, i-j);

                  # remove leading zeros.
```

```
                    if i = j then
                       l:= 0;
                       while gamma[l+1] = l do
                          l:= l+1;
                       od;
                       gamma:= Sublist(gamma, [l+1..Length(gamma)]) - l;
                    fi;

                    Add(hks[j], sign * Position(pm[m-j], gamma));
                 fi;
              fi;
           od;

        od;
        return hks;
     end;

     # collect hook encodings.
     for i in [1..n] do
        pm[i]:= List(Partitions(i), BetaSet);
        scheme[i]:= [];
        for beta in pm[i] do
           Add(scheme[i], hooks(beta, i));
        od;
     od;

     return scheme;

  end;
```

Let t denote the column of ρ in the character table of S_{n-k}. Then the Murnaghan–Nakayama formula can be restated in the following efficient way.

$$\chi^{\alpha}(\pi) = \sum_{f \in \texttt{scheme[n][i][k]}} \mathrm{sgn}(f) t[\mathrm{abs}(f)].$$

Now the matrix of character values can be constructed by exactly the same algorithm as the one we used in section 1 for the partitions of n. Just that adding of a cycle to known partitions is replaced by computing a new column of a character table according to the last version of the Murnaghan–Nakayama formula. This is done by the local function `charCol` which constructs a new column for S_m from the known column `t` of S_{m-k} by looking for k-hooks.

```
  IrrS:= function(n)

     local scheme, pm, i, m, k, t, col, np, res, charCol;

     scheme:= InductionScheme(n);

     # how to construct a new column.
     charCol:= function(m, t, k)
        local i, col, pi, val;

        col:= [];
        for pi in scheme[m] do
           val:= 0;
           for i in pi[k] do
```

```
            if i < 0 then
               val:= val - t[-i];
            else
               val:= val + t[i];
            fi;
         od;
         Add(col, val);
      od;
      return col;
   end;

   #  construct the columns.
   pm:= List([1..n-1], x-> []);
   for m in [1..QuoInt(n,2)] do
      Add(pm[m], charCol(m, [1], m));

      for k in [m+1..n-m] do
         for t in pm[k-m] do
            Add(pm[k], charCol(k, t, m));
         od;
      od;
   od;

   #  collect and transpose.
   np:= Length(scheme[n]);
   res:= List([1..np], x-> []);
   for k in [1..n-1] do
      for t in pm[n-k] do
         col:= charCol(n, t, k);
         for i in [1..np] do
            Add(res[i], col[i]);
         od;
      od;
   od;

   col:= charCol(n, [1], n);
   for i in [1..np] do
      Add(res[i], col[i]);
   od;

   return res;

end;
```

With this method we can compute really big character tables in a short time as long as there is enough storage memory available on the computer. The following times were achieved in an 8 megabyte GAP. (S_{20} already has 627 conjugacy classes and characters.)

```
gap> IrrS(8);; time;
429
gap> IrrS(20);; time;
163770
```

This function `IrrS` is installed in the field `matrix` of the record `sym` where the function `CharTableSpecialized` will find and use it rather than to compute every single character value with the function `chi3` if the whole character table is to be constructed.

```
    sym.matrix:= IrrS;;
```

Try to figure out how these techniques can be applied to the computation of the character table of alternating groups.

4 Case B: Wreath Products.

The Weyl group of type B_n can be described as the wreath product of a cyclic group of order 2 with the symmetric group S_n. Both the classes and the characters of B_n are parametrized by pairs of partitions.

We will solve the problem of the computation of the character tables of these groups in the more general context of wreath products of arbitrary groups with symmetric groups.

Let G be a finite group with r conjugacy classes represented by $(g_1, \ldots, g_r)$ and character table $\Phi = (\phi_i(g_j))$. Denote by X_n the wreath product of G with the symmetric group S_n. The elements of X_n are of the form $(f; \gamma) = (f_1, \ldots, f_n; \gamma)$ with $f_i \in G$ for $i = 1, \ldots, n$ and $\gamma \in S_n$. For such an element π of X_n and a k-cycle $\kappa = (j, j\kappa, \ldots, j\kappa^{k-1})$ denote by

$$g((f; \gamma), \kappa) := f_j f_{j\kappa^{-1}} f_{j\kappa^{-2}} \cdots f_{j\kappa^{-(k-1)}}$$

the **cycle product** of π and κ. Note that $g(\pi, \kappa) \in G$. Let π be the cycle structure of γ as defined in section 1. This partition π will now be split into r partitions $(\pi^1, \ldots, \pi^r)$ according to the following rule. If the cycle product $g((f; \gamma), \kappa)$ is conjugate to g_i in G then its corresponding part in π will belong to π^i. This procedure will result in an **r-tuple of partitions of n** which describes the **cycle structure** of $(f; \gamma)$. Now the same statement as is section 1 holds.

Proposition 4.1 ([4], (4.2.8)) *Two elements of X_n are conjugate if and only if they have the same cycle structure.*

Note that on the other hand for every r-tuple of partitions of n there is an element of X_n with that cycle structure. Thus the classes of X_n are parametrized by the r-tuples of partitions of n. The function `PartitionTuples` that constructs the set of r-tuples of partitions of n is composed in the same way as the function `partitions`. This time we have to be careful about the positions in the r-tuple where an m-cycle is to be inserted. To the partition which consists of only the m-part now correspond all those r-tuples of partitions which have an m-part in one place and the empty partition in all other places. Moreover for each m we have to keep track of a position where the last m-cycle has been added to a tuple. Only in this place and in higher positions more m-cycles are allowed. So we will construct the auxiliary tuples as records with fields for the actual tuple and the position information. Only in the last step where we collect these tuples and add a k-cycle we will omit the position field.

```
    PartitionTuples:= function(n, r)
```

```
local m, k, pm, i, t, t1, s, res, empty;

# the empty partition tuple.
empty:= rec(
   tup:= List([1..r], x-> []),
   pos:= List([1..n-1], x-> 1));

# trivial case.
if n = 0 then
   return [empty.tup];
fi;

pm:= List([1..n-1], x-> []);
for m in [1..QuoInt(n, 2)] do

   # the m-cycle in all possible places.
   for i in [1..r] do
      s:= Copy(empty);
      s.tup[i]:= [m];
      s.pos[m]:= i;
      Add(pm[m], s);
   od;

   # add the m-cycle to everything you know.
   for k in [m+1..n-m] do
      for t in pm[k-m] do
         for i in [t.pos[m]..r] do
            t1:= Copy(t);
            s:= [m];
            Append(s, t.tup[i]);
            t1.tup[i]:= s;
            t1.pos[m]:= i;
            Add(pm[k], t1);
         od;
      od;
   od;
od;

# collect.
res:= [];
for k in [1..n-1] do
   for t in pm[n-k] do
      for i in [t.pos[k]..r] do
         t1:= Copy(t.tup);
         s:= [k];
         Append(s, t.tup[i]);
         t1[i]:= s;
         Add(res, t1);
      od;
   od;
od;

# finally the n-cycle.
for i in [1..r] do
   s:= Copy(empty.tup);
   s[i]:= [n];
   Add(res, s);
od;
```

```
      # return the result.
      return res;

   end;
```

The size of the centralizer is best stated in terms of cycles of equal length. Let $x \in X_n$ have cycle structure $\pi = (\pi^1, \ldots, \pi^r)$ and let $a_{ik}(x)$ be the number of k-parts of π^i. The matrix $a(x) := (a_{ik}(x))$ is called the **type** of x. The order of the centralizer of an element in X_n depends on the orders of the centralizers in G and is given by the following formula.

Lemma 4.2 ([4], (4.2.10)) *Let $x \in X_n$ be of type $a(x)$. Then*

$$|C_{X_n}(x)| = \prod_{i,k} a_{ik}(x)!\,(k|C_G(g_i)|)^{a_{ik}(x)}\,.$$

The corresponding GAP function, however, deals with r-tuples of partitions and is a generalization of the function which computes the sizes of centralizers in section 1. Here the parameter sub_cen is the list of the orders of the centralizers in G.

```
   CentralizerWreath := function(sub_cen, ptuple)

      local p, i, j, k, last, res;

      res:= 1;
      for i in [1..Length(ptuple)] do
         last:= 0; k:= 1;
         for p in ptuple[i] do
            res:= res * sub_cen[i] * p;
            if p = last then
               k:= k+1;
               res:= res * k;
            else
               k:= 1;
            fi;
            last:= p;
         od;
      od;

      return res;

   end;
```

The powermaps of the wreath product X_n depend on the powermaps of G. They can be described cycle by cycle. If the length k of a cycle in π^i is divisible by the prime p then this cycle is replaced by p copies of k/p. Otherwise this cycle is moved to π^j where g_i^p is conjugate to g_j in G.

The function PowerWreath computes the r-tuple of partitions corresponding to the p-th power of an element with cycle structure *ptuple*. Here sub_pm is assumed to be the p-th powermap of the group G.

```
   PowerWreath := function(sub_pm, ptuple, p)

      local power, k, i, j;
```

```
    power:= List(ptuple, x-> []);
    for i in [1..Length(ptuple)] do
       for k in ptuple[i] do
          if k mod p = 0 then
             for j in [1..p] do
                Add(power[i], k/p);
             od;
          else
             Add(power[sub_pm[i]], k);
          fi;
       od;
    od;

    for k in power do
       Sort(k, function(a,b) return a>b; end);
    od;
    return power;

 end;
```

The r-tuples of partitions are also used as labels for the irreducible characters of the wreath product X_n. Let $\alpha = (\alpha^1, \dots, \alpha^r)$ be an r-tuple of partitions of n. The irreducible character χ^α belonging to this r-tuple of partitions can be described as a tensor product.

For every $1 \le c \le r$ we take $|\alpha^c|$ copies of the irreducible character ϕ_c of G. The product

$$\prod_{c=1}^{r} \phi_c^{\alpha^c}$$

of these characters is a character of the normal subgroup G^n of X_n which can in a natural way be extended to its inertia group

$$T_\alpha = T_{\alpha_1} \times \cdots \times T_{\alpha_r} \cong X_{|\alpha^1|} \times \cdots \times X_{|\alpha^r|}$$

giving the irreducible character

$$\prod_{c=1}^{r} \overline{\phi_c^{\alpha^c}}$$

of T_α. By Clifford theory all irreducible characters of T_α extending $\prod_{c=1}^{r} \phi_c^{\alpha^c}$ are obtained as tensor products with irreducible characters of the inertia factor

$$T_\alpha / G^n \cong S_{|\alpha_1|} \times \cdots \times S_{|\alpha_r|}.$$

Inducing such a character up to X_n yields an irreducible character of X_n. Moreover every irreducible character of X_n is of the form

$$\chi^\alpha = \left(\prod_{c=1}^{r} \overline{\phi_c^{\alpha^c}} \otimes \chi^{\alpha^c} \right)^{X_n}.$$

Proposition 4.3 ([4], (4.4.3)) *The characters χ^α run through a complete system of pairwise different and irreducible characters of X_n if α runs through all r–tuples of partitions of n.*

The following generalization of the Murnaghan–Nakayama formula allows the evaluation of the character χ^α of X_n on the element π described by its type and thereby establishes the relationship between classes and characters of X_n in terms of r-tuples of partitions of n.

Theorem 4.4 *Let $\alpha = (\alpha^1, \ldots, \alpha^r)$ with $\sum_i |\alpha^i| = n$ be an r–tuple of partitions of n and let $\pi \in X_n$ with $a_{tk}(\pi) > 0$ for fixed $k \leq n$, $t \leq r$. Let $\rho \in X_{n-k}$ be of type $a(\rho)$ where*

$$a_{ij}(\rho) = \begin{cases} a_{tk}(\pi) - 1 & \text{if } i = t,\ j = k, \\ a_{ij}(\pi) & \text{otherwise.} \end{cases}$$

Define $\chi^{[\ ]} = 1$. Then

$$\chi^\alpha(\pi) = \sum_{s=1}^{r} \phi_s(g_t) \sum_{h_{ij}^{\alpha^s} = k} (-1)^{l_{ij}^{\alpha^s}} \chi^{\alpha - R_{ij}^{\alpha^s}}(\rho).$$

Proof. This formula is proved by a direct computation which first reveals the place where the ordinary Murnaghan–Nakayama formula (1.4) can be applied and then collects the results together to give the desired formula. Write χ^α as an induced character,

$$\chi^\alpha(\pi) = \sum_\sigma \prod_c \left(\overline{\phi_c^{|\alpha^c|}} \otimes \chi^{\alpha^c} \right) (\pi^\sigma)$$

where the sum ranges over all coset representatives σ in X_n/T_α with $\pi^\sigma \in T_\alpha$. The coset representatives σ can be chosen as elements of the complement S_n of G^n in X_n. Then write $\pi = \kappa\rho$ for a suitable ρ which satisfies the hypothesis of the theorem and split the sum into parts according to whether κ^σ lies in the direct factor T_{α^s} of the inertia factor T_α. Extracting the factor of the character corresponding to T_{α^s} we obtain

$$\chi^\alpha(\pi) = \sum_{s=1}^{r} \sum_\sigma \overline{\phi_s^{|\alpha^s|}} \chi^{\alpha^s}((\kappa\rho_1)^\sigma) \prod_{c \neq s} \overline{\phi_c^{|\alpha^c|}} \chi^{\alpha^c}(\rho_2^\sigma)$$

where σ runs over those coset representatives which satisfy $\pi^\sigma \in T_\alpha$ and $\kappa^\sigma \in T_{\alpha^s}$ and where $\rho = \rho_1\rho_2$ such that $(\kappa\rho_1)^\sigma$ is the projection of π^σ on T_{α^s}.

The extracted factor $\overline{\phi_s^{|\alpha^s|}} \chi^{\alpha^s}((\kappa\rho_1)^\sigma)$ can be evaluated as follows. First observe that

$$\overline{\phi_s^{|\alpha^s|}}((\kappa\rho_1)^\sigma) = \overline{\phi_s^k}(\kappa^\sigma) \overline{\phi_s^{|\alpha^s|-k}}(\rho_1^\sigma) = \phi_s(g_t) \overline{\phi_s^{|\alpha^s|-k}}(\rho_1^\sigma)$$

by ([4], (4.3.9)) and the choice of κ. The remaining part of the factor is essentially a character of the symmetric group $S_{|\alpha^s|}$ and the Murnaghan-Nakayama formula 1.4 can be applied,

$$\chi^{\alpha^s}((\kappa\rho_1)^\sigma) = \sum_{h_{ij}^{\alpha^s}=k} (-1)^{l_{ij}^{\alpha^s}} \chi^{\alpha - R_{ij}^{\alpha^s}}(\rho_1).$$

Putting the results together again we have

$$\chi^\alpha(\pi) = \sum_{s=1}^{r} \phi_s(g_t) \sum_{h_{ij}^{\alpha^s}=k} (-1)^{l_{ij}^{\alpha^s}} \sum_\sigma \prod_c \left(\overline{\phi_c^{|\alpha_\circ^c|}} \otimes \chi^{\alpha_\circ^c} \right)(\rho^\sigma)$$

for a sum over certain coset representatives σ which defines the induced character $\chi^{\alpha_\circ}(\rho)$ and $\alpha_\circ = \alpha - R_{ij}^{\alpha^s}$. This observation completes the proof. $\square$

An implementation of this general formula will be given on page 37. Before that we will install the generic character table for Weyl groups of type B.

The Weyl group of type B_n is a special case of a wreath product with a symmetric group. It is isomorphic to the wreath product of a cyclic group of order 2 with a symmetric group. If we take the known fixed values from the character table of the cyclic group of order 2 we can implement a generic table `wlb` without any further dependencies.

```
wlb:= rec(name:= "WeylB");
wlb.order:= (n-> 2^n * Factorial(n));
wlb.specializedname:= (n-> ConcatenationString("W(B",String(n),")"));
wlb.text:= "generic character table for Weyl groups of type B";
wlb.isGenericTable:= true;
wlb.domain:= (n-> IsInt(n) and n > 0);
```

As before we install the lists of other functions.

```
wlb.classparam:= []; wlb.charparam:= []; wlb.centralizers:= [];
wlb.orders:= []; wlb.powermap:= []; wlb.irreducibles:= [[]];
```

The classes and the characters are labelled by pairs of partitions.

```
wlb.classparam[1]:= function(n)
   return PartitionTuples(n, 2);
end;

wlb.charparam[1]:= function(n)
   return PartitionTuples(n, 2);
end;
```

The orders of the elements are computed from the labels in the following way (s. [4], (4.2.12)).

```
wlb.orders[1]:= function(n, lbl)
   local ord;

   ord:= 1;
   if lbl[1] <> [] then
      ord:= Lcm(lbl[1]);
   fi;
   if lbl[2] <> [] then
```

```
      ord:= Lcm(ord, 2 * Lcm(lbl[2]));
   fi;

   return ord;

end;
```

The centralizers are determined by the function `CentralizerWreath`.

```
wlb.centralizers[1]:= function(n, lbl)
   return CentralizerWreath([2, 2], lbl);
end;
```

The function which constructs the powermap has to distinguish two cases, depending on whether the prime is odd. If the prime p equals 2 then `PowerWreath` is called with the second powermap of the cyclic group of order 2, otherwise with the identity mapping

```
wlb.powermap[1]:= function(n, lbl, p)
   if p = 2 then
      return [1, PowerWreath([1, 1], lbl, 2)];
   else
      return [1, PowerWreath([1, 2], lbl, p)];
   fi;
end;
```

As in section 1 the Murnaghan–Nakayama formula enables a straightforward implementation `psi1` of the character values.

```
psi1:= function(n, alpha, pi)

   local i, j, k, s, t, delta, rho, res;

   # termination condition.
   if n = 0  then
      return 1;
   fi;

   # negative cycles first.
   if pi[2] <> [] then
      t:= 2;
   else
      t:= 1;
   fi;

   # get length of longest cycle.
   k:= pi[t][1];

   # construct rho.
   rho:= Copy(pi);
   rho[t]:= Sublist(pi[t], [2..Length(pi[t])]);

   res:= 0;

   # loop over the young diagram.
   for s in [1, 2] do
      for i in [1..Length(alpha[s])] do
         for j in [1..alpha[s][i]] do
            if hooklength(alpha[s], i, j) = k then
```

```
                delta:= Copy(alpha);
                delta[s]:= unrimmed(alpha[s], i, j);

                # enter recursion.
                if s = 2 and t = 2 then
                   res:= res - (-1)^leglength(alpha[s], i, j)
                        * psi1(n-k, delta, rho);
                else
                   res:= res + (-1)^leglength(alpha[s], i, j)
                        * psi1(n-k, delta, rho);
                fi;
             fi;
          od;
       od;
    od;

    # return the result.
    return res;

end;

wlb.irreducibles[1][1]:= psi1;
```

Once again a generic character table is complete and we can compute as a small example the character table of the Weyl group of type B_3.

```
gap> b3:= CharTableSpecialized(wlb, 3);
rec( name := "W(B3)", order := 48, centralizers :=
[ 48, 16, 16, 48, 8, 8, 8, 8, 6, 6 ], orders :=
[ 1, 2, 2, 2, 2, 4, 2, 4, 3, 6 ], powermap :=
[ , [ 1, 1, 1, 1, 1, 3, 1, 3, 9, 9 ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 1, 4 ] ], irreducibles :=
[ [ 1, 1, 1, 1, -1, -1, -1, -1, 1, 1 ],
  [ 3, 1, -1, -3, -1, -1, 1, 1, 0, 0 ],
  [ 3, -1, -1, 3, -1, 1, -1, 1, 0, 0 ],
  [ 1, -1, 1, -1, -1, 1, 1, -1, 1, -1 ],
  [ 2, 2, 2, 2, 0, 0, 0, 0, -1, -1 ],
  [ 3, -1, -1, 3, 1, -1, 1, -1, 0, 0 ],
  [ 3, 1, -1, -3, 1, 1, -1, -1, 0, 0 ],
  [ 2, -2, 2, -2, 0, 0, 0, 0, -1, 1 ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 1, -1, 1, -1, 1, -1, -1, 1, 1, -1 ] ], classparam :=
[ [ 1, [ [ 1, 1, 1 ], [  ] ] ], [ 1, [ [ 1, 1 ], [ 1 ] ] ],
  [ 1, [ [ 1 ], [ 1, 1 ] ] ], [ 1, [ [  ], [ 1, 1, 1 ] ] ],
  [ 1, [ [ 2, 1 ], [  ] ] ], [ 1, [ [ 1 ], [ 2 ] ] ],
  [ 1, [ [ 2 ], [ 1 ] ] ], [ 1, [ [  ], [ 2, 1 ] ] ],
  [ 1, [ [ 3 ], [  ] ] ], [ 1, [ [  ], [ 3 ] ] ] ], irredinfo :=
[ rec(
      charparam := [ 1, [ [ 1, 1, 1 ], [  ] ] ] ), rec(
      charparam := [ 1, [ [ 1, 1 ], [ 1 ] ] ] ), rec(
      charparam := [ 1, [ [ 1 ], [ 1, 1 ] ] ] ), rec(
      charparam := [ 1, [ [  ], [ 1, 1, 1 ] ] ] ), rec(
      charparam := [ 1, [ [ 2, 1 ], [  ] ] ] ), rec(
      charparam := [ 1, [ [ 1 ], [ 2 ] ] ] ), rec(
      charparam := [ 1, [ [ 2 ], [ 1 ] ] ] ), rec(
      charparam := [ 1, [ [  ], [ 2, 1 ] ] ] ), rec(
      charparam := [ 1, [ [ 3 ], [  ] ] ] ), rec(
      charparam := [ 1, [ [  ], [ 3 ] ] ] )
 ], text := "computed using generic character table for Weyl groups\
 of type B", classes := [ 1, 3, 3, 1, 6, 6, 6, 6, 8, 8
 ], operations := CharTableOps )
```

As usual the new table is checked for consistency and displayed.

```
gap> TestCharTable(b3);
true
gap> DisplayCharTable(b3);
W(B3)

    2  4  4  4  4  3  3  3  3  1  1
    3  1  .  .  1  .  .  .  .  1  1

      1a 2a 2b 2c 2d 4a 2e 4b 3a 6a
   2P 1a 1a 1a 1a 1a 2b 1a 2b 3a 3a
   3P 1a 2a 2b 2c 2d 4a 2e 4b 1a 2c

X.1    1  1  1  1 -1 -1 -1 -1  1  1
X.2    3  1 -1 -3 -1 -1  1  1  .  .
X.3    3 -1 -1  3 -1  1 -1  1  .  .
X.4    1 -1  1 -1 -1  1  1 -1  1 -1
X.5    2  2  2  2  .  .  .  . -1 -1
X.6    3 -1 -1  3  1 -1  1 -1  .  .
X.7    3  1 -1 -3  1  1 -1 -1  .  .
X.8    2 -2  2 -2  .  .  .  . -1  1
X.9    1  1  1  1  1  1  1  1  1  1
X.10   1 -1  1 -1  1 -1 -1  1  1 -1
```

The construction of the character table of the Weyl group of type B_6 with this generic table needs roughly two minutes. This group has 65 classes.

```
gap> b6:= CharTableSpecialized(wlb, 6);; time;
114833
```

The function `psi1` can be improved by using better termination conditions similar to those in `phi2` in section 1. We will not do that but present now a version which deals with pairs of beta sets instead of pairs of partitions.

```
psi2:= function(n, beta, pi)

   local i, j, k, lb, o, s, t, gamma, rho, sign, val;

   # termination condition.
   if n = 0  then
      return 1;
   fi;

   # negative cycles first.
   if pi[2] <> [] then
      t:= 2;
   else
      t:= 1;
   fi;

   # get length of longest cycle.
   k:= pi[t][1];

   # construct rho.
   rho:= Copy(pi);
```

```
      rho[t]:= Sublist(pi[t], [2..Length(pi[t])]);

      val:= 0;

      # loop over the beta sets.
      for s in [1, 2] do

         # determine offset.
         o:= 0;
         lb:= Length(beta[s]);
         while o < lb and beta[s][o+1] = o do
            o:= o+1;
         od;

         for i in beta[s] do
            if  i >= k+o and not i-k in beta[s]  then

               # compute the leg parity.
               sign:= 1;
               for j in [i-k+1..i-1] do
                  if j in beta[s] then
                      sign:= -sign;
                  fi;
               od;

               # consider character table of C2.
               if  s = 2 and t = 2  then
                  sign:= -sign;
               fi;

               # construct new beta set.
               gamma:= Copy(beta);
               SubtractSet(gamma[s], [i]);
               AddSet(gamma[s], i-k);

               # enter recursion.
               val:= val + sign * psi2(n-k, gamma, rho);
            fi;
         od;
      od;

      # return the result.
      return val;

   end;

   wlb.irreducibles[1][1]:= function(n, alpha, pi)
      return psi2(n, [BetaSet(alpha[1]), BetaSet(alpha[2])], pi);
   end;
```

The construction of the character table of the Weyl group of type B_6 now is slightly faster.

```
gap> b6:= CharTableSpecialized(wlb, 6);; time;
90857
```

The record `wlb` we have just defined is called `CharTableWeylB` in the GAP library. It will be used by the function `CharTable` if it is called with `"WeylB"` as a first argument.

Only slight changes are necessary to obtain from the function `psi2` the more general function `CharValueWreathSymmetric` which computes a single character value of a wreath product.

```
CharValueWreathSymmetric := function(sub, n, beta, pi)

   local i, j, k, lb, o, s, t, r, gamma, rho, sign, val;

   # termination condition.
   if n = 0  then
      return 1;
   fi;

   r:= Length(pi);

   # negative cycles first.
   t:= r;
   while pi[t] = [] do
      t:= t-1;
   od;

   # get length of longest cycle.
   k:= pi[t][1];

   # construct rho.
   rho:= Copy(pi);
   rho[t]:= Sublist(pi[t], [2..Length(pi[t])]);

   val:= 0;

   # loop over the beta sets.
   for s in [1..r] do

      # determine offset.
      o:= 0;
      lb:= Length(beta[s]);
      while o < lb and beta[s][o+1] = o do
         o:= o+1;
      od;

      for i in beta[s] do
         if  i >= k+o and not i-k in beta[s]  then

            # compute the leg parity.
            sign:= 1;
            for j in [i-k+1..i-1] do
               if j in beta[s] then
                   sign:= -sign;
               fi;
            od;

            # consider character table of subgroup.
            sign:= sub.irreducibles[s][t] * sign;

            # construct new beta set.
            gamma:= Copy(beta);
            SubtractSet(gamma[s], [i]);
            AddSet(gamma[s], i-k);
            # enter recursion.
            val:= val +
                 sign * CharValueWreathSymmetric(sub, n-k, gamma, rho);
         fi;
      od;
   od;

   # return the result.
   return val;

end;
```

Along the lines of `PartitionTuples` one can now write a function `IrrX` which constructs the character table of a wreath product with a symmetric group in a recursion free fashion.

```
IrrX:= function(tbl, n)

   local i, j, k, m, r, s, t, pm, res, col, scheme, np, charCol, hooks,
     pts, partitions;

   r:= Length(tbl.irreducibles[1]);

   # encode partition tuples by positions of partitions.
   partitions:= List([1..n], Partitions);
   pts:= [];
   for i in [1..n] do
      pts[i]:= PartitionTuples(i, r);
      for j in [1..Length(pts[i])] do
         np:= [[], []];
         for t in pts[i][j] do
            s:= Sum(t);
            Add(np[1], s);
            if s = 0 then
               Add(np[2], 1);
            else
               Add(np[2], Position(partitions[s], t));
            fi;
         od;
         pts[i][j]:= np;
      od;
   od;

   scheme:= InductionScheme(n);

   # how to encode a hook.
   hooks:= function(np, n)
      local res, i, k, l, ni, pi, sign;

      res:= [];
      for i in [1..r] do
         res[i]:= List([1..n], x-> []);
      od;

      for i in [1..r] do
         ni:= np[1][i]; pi:= np[2][i];
         for k in [1..ni] do
            for l in scheme[ni][pi][k] do
               np[1][i]:= ni-k;
               if l < 0 then
                  np[2][i]:= -l;
                  sign:= -1;
               else
                  np[2][i]:= l;
                  sign:= 1;
               fi;
               if k = n then
                  Add(res[i][k], sign);
               else
                  Add(res[i][k], sign * Position(pts[n-k], np));
               fi;
            od;
         od;
         np[1][i]:= ni; np[2][i]:= pi;
      od;
      return res;
   end;

   # collect hook encodings.
   res:= [];
   for i in [1..n] do
      res[i]:= [];
      for np in pts[i] do
         Add(res[i], hooks(np, i));
      od;
   od;
   scheme:= res;

   # how to construct a new column.
   charCol:= function(n, t, k, p)
      local i, j, col, pi, val;

      col:= [];
      for pi in scheme[n] do
         val:= 0;
         for j in [1..r] do
            for i in pi[j][k] do
               if i < 0 then
                  val:= val - tbl.irreducibles[j][p] * t[-i];
               else
                  val:= val + tbl.irreducibles[j][p] * t[i];
               fi;
            od;
         od;
         Add(col, val);
      od;
      return col;
   end;

   # construct the columns.
   pm:= List([1..n-1], x->[]);
   for m in [1..QuoInt(n,2)] do

      # the m-cycle in all possible places
      for i in [1..r] do
         s:= [1..n]*0+1;
         s[m]:= i;
         Add(pm[m], rec(col:= charCol(m, [1], m, i), pos:= s));
```

```
        od;

        # add the $m$-cycle to everything you know
        for k in [m+1..n-m] do
           for t in pm[k-m] do
              for i in [t.pos[m]..r] do
                 s:= Copy(t.pos);
                 s[m]:= i;
                 Add(pm[k], rec(col:= charCol(k, t.col, m, i), pos:= s));
              od;
           od;
        od;
     od;

     # collect and transpose.
     np:= Length(scheme[n]);
     res:= List([1..np], x-> []);
     for k in [1..n-1] do
        for t in pm[n-k] do
           for i in [t.pos[k]..r] do
              col:= charCol(n, t.col, k, i);
              for j in [1..np] do
                 Add(res[j], col[j]);
              od;
           od;
        od;
     od;

     for i in [1..r] do
        col:= charCol(n, [1], n, i);
        for j in [1..np] do
           Add(res[j], col[j]);
        od;
     od;

     return res;

  end;
```

IrrX is called MatCharsWreathSymmetric in the GAP library. It can also be used for a fast computation of the matrix of character values in wlb. We only have to give as a first argument a record which contains in the field irreducibles the matrix of character values of the cyclic group of order 2.

```
  wlb.matrix:= (n-> IrrX(rec(irreducibles:= [[1, 1], [1, -1]]), n));
```

Now the character table of the Weyl group of type B_6 can be constructed in three seconds.

```
  gap> b6:= CharTableSpecialized(wlb, 6);; time;
  3004
```

This section ends with the function CharTableWreathSymmetric which computes the character table of a wreath product of a group with character table sub with a symmetric group on n points.

```
  CharTableWreathSymmetric := function(sub, n)

     local i, j, tbl, r, nccl, parts, p, pm, spm;
```

```
   # initialize.
   tbl:= rec(
      order:= sub.order^n * Factorial(n),
      name:= ConcatenationString(sub.name, "wrS", String(n)),
      centralizers:= [],
      classes:= [],
      orders:= [],
      powermap:= [],
      operations:= CharTableOps);

   # the table head.
   r:= Length(sub.orders);
   parts:= PartitionTuples(n, r);
   tbl.classparam:= parts;
   tbl.irredinfo:= [];
   nccl:= Length(parts);
   for i in [1..nccl] do
      tbl.centralizers[i]:=
          CentralizerWreath(sub.centralizers, parts[i]);
      tbl.classes[i]:= tbl.order / tbl.centralizers[i];
      pm:= 1;
      for j in [1..r] do
         if parts[i][j] <> [] then
            pm:= Lcm(pm, sub.orders[j] * Lcm(parts[i][j]));
         fi;
      od;
      tbl.orders[i]:= pm;
      tbl.irredinfo[i]:= rec(charparam:= parts[i]);
   od;

   # the powermap.
   for p in Set(Factors(tbl.order)) do
      pm:= [];
      if  IsBound(sub.powermap[p]) then
          spm:= sub.powermap[p];
      else
          spm:= Parametrized(Powermap(sub, p, rec(quick:=true)));
      fi;
      for i in [1..nccl] do
         pm[i]:= Position(parts, PowerWreath(spm, parts[i], p));
      od;
      tbl.powermap[p]:= pm;
   od;

   # the character values.
   tbl.irreducibles:= IrrX(sub, n);

   return tbl;

end;
```

With this function we can for example compute the character table of the wreath product $S_3 \wr S_2$.

```
gap> s3:= CharTableSpecialized(sym, 3);;
gap> s3wrs2:= CharTableWreathSymmetric(s3, 2);
rec( order := 72, name := "S3wrS2", centralizers :=
[ 72, 12, 18, 8, 6, 18, 12, 4, 6 ], classes :=
[ 1, 6, 4, 9, 12, 4, 6, 18, 12 ], orders :=
[ 1, 2, 3, 2, 6, 3, 2, 4, 6 ], powermap :=
[ , [ 1, 1, 3, 1, 3, 6, 1, 4, 6 ], [ 1, 2, 1, 4, 2, 1, 7, 8, 7 ]
 ], operations := CharTableOps, classparam :=
[ [ [ 1, 1 ], [  ], [  ] ], [ [ 1 ], [ 1 ], [  ] ],
  [ [ 1 ], [  ], [ 1 ] ], [ [  ], [ 1, 1 ], [  ] ],
  [ [  ], [ 1 ], [ 1 ] ], [ [  ], [  ], [ 1, 1 ] ],
  [ [ 2 ], [  ], [  ] ], [ [  ], [ 2 ], [  ] ],
  [ [  ], [  ], [ 2 ] ] ], irredinfo := [ rec(
      charparam := [ [ 1, 1 ], [  ], [  ] ] ), rec(
      charparam := [ [ 1 ], [ 1 ], [  ] ] ), rec(
      charparam := [ [ 1 ], [  ], [ 1 ] ] ), rec(
      charparam := [ [  ], [ 1, 1 ], [  ] ] ), rec(
      charparam := [ [  ], [ 1 ], [ 1 ] ] ), rec(
      charparam := [ [  ], [  ], [ 1, 1 ] ] ), rec(
      charparam := [ [ 2 ], [  ], [  ] ] ), rec(
      charparam := [ [  ], [ 2 ], [  ] ] ), rec(
      charparam := [ [  ], [  ], [ 2 ] ] ) ], irreducibles :=
[ [ 1, -1, 1, 1, -1, 1, -1, 1, -1 ],
  [ 4, -2, 1, 0, 1, -2, 0, 0, 0 ], [ 2, 0, 2, -2, 0, 2, 0, 0, 0 ],
  [ 4, 0, -2, 0, 0, 1, -2, 0, 1 ], [ 4, 2, 1, 0, -1, -2, 0, 0, 0 ],
  [ 1, 1, 1, 1, 1, 1, -1, -1, -1 ],
  [ 1, -1, 1, 1, -1, 1, 1, -1, 1 ],
  [ 4, 0, -2, 0, 0, 1, 2, 0, -1 ], [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ]
 ] )
gap> TestCharTable(s3wrs2);
true
gap> DisplayCharTable(s3wrs2);
S3wrS2

     2  3  2  1  3  1  1  2  2  1
     3  2  1  2  .  1  2  1  .  1

       1a 2a 3a 2b 6a 3b 2c 4a 6b
    2P 1a 1a 3a 1a 3a 3b 1a 2b 3b
    3P 1a 2a 1a 2b 2a 1a 2c 4a 2c

X.1     1 -1  1  1 -1  1 -1  1 -1
X.2     4 -2  1  .  1 -2  .  .  .
X.3     2  .  2 -2  .  2  .  .  .
X.4     4  . -2  .  .  1 -2  .  1
X.5     4  2  1  . -1 -2  .  .  .
X.6     1  1  1  1  1  1 -1 -1 -1
X.7     1 -1  1  1 -1  1  1 -1  1
X.8     4  . -2  .  .  1  2  . -1
X.9     1  1  1  1  1  1  1  1  1
```

The first argument, however, can be any character table not necessarily one of a symmetric group.

5 Case D.

The Weyl group of type D_n is a normal subgroup of index two in the Weyl group of type B_n. As in the case of the alternating group its character table can be derived from that of the Weyl group of type B_n by mainly using Clifford theory. We will therefore perform the same steps as in section 2 to define a generic character table `wld` for Weyl groups of type D.

We begin as usual by initializing the record `wld`.

```
wld:= rec(name:= "WeylD");
wld.order:= (n-> 2^(n-1) * Factorial(n));
wld.specializedname:= (n-> ConcatenationString("W(D",String(n),")"));
wld.text:= "generic character table for Weyl groups of type D";
wld.isGenericTable:= true;
wld.domain:= (n-> IsInt(n) and n > 0);
wld.classparam:= []; wld.charparam:= []; wld.centralizers:= [];
wld.orders:= []; wld.powermap:= []; wld.irreducibles:= [[]];
```

The classes of a Weyl group of type D are labelled by pairs of partitions like in case B. Classes of B_n belonging to D_n are exactly those with label (π_1, π_2) where π_2 has an even length. (Here we assume that π_2 corresponds to the element of order two in the cyclic group of order two.) A class (π_1, π_2) of B_n will split into two classes of equal length in D_n if and only if π_2 is the empty partition and π_1 consists of only even parts (i.e. $\pi_1 = 2 \cdot \pi$ for some partition π of $n/2$). You see that this can only happen if n is even. In that case we will write a sign instead of the empty partition for π_2 in order to distinguish the two classes of D_n. So the class parameters of D_n are obtained as follows.

```
wld.classparam[1]:= function(n)

   local classes, pi;

   classes:= [];
   for pi in PartitionTuples(n, 2) do
      if Length(pi[2]) mod 2 = 0 then
         if pi[2] = [] and ForAll(pi[1], x-> x mod 2 = 0) then
            Add(classes, [pi[1], '+']);
            Add(classes, [pi[1], '-']);
         else
            Add(classes, pi);
         fi;
      fi;
   od;

   return classes;

end;
```

The orders of the elements in D_n are of course the same as in B_n but we have to take care of the parametrization of the split classes.

```
wld.orders[1]:= function(n, lbl)

   local ord;

   ord:= 1;
   if lbl[1] <> [] then
      ord:= Lcm(lbl[1]);
   fi;
   if lbl[2] <> [] and IsList(lbl[2]) then
      ord:= Lcm(ord, 2*Lcm(lbl[2]));
   fi;
```

```
        return ord;

    end;
```

The orders of the centralizers in D_n are half of the orders of the centralizers in B_n except for the split classes.

```
    wld.centralizers[1]:= function(n, lbl)
        if not IsList(lbl[2]) then
            return CentralizerWreath([2,2], [lbl[1], []]);
        else
            return CentralizerWreath([2,2], lbl) / 2;
        fi;
    end;
```

Again the powermap has to distinguish two cases. Moreover it has to handle the special labelling of the split classes. If a power of a split class is again a split class we decide to keep the sign of the label.

```
    wld.powermap[1]:= function(n, lbl, p)

        local power;

        if not IsList(lbl[2]) then
            power:= PowerPartition(lbl[1], p);
            if ForAll(power, x-> x mod 2 = 0) then
                return [1, [power, lbl[2]]];   # keep the sign.
            else
                return [1, [power, []]];
            fi;
        else
            if p = 2 then
                return [1, PowerWreath([1, 1], lbl, 2)];
            else
                return [1, PowerWreath([1, 2], lbl, p)];
            fi;
         fi;
    end;
```

The characters $\chi^{(\alpha_1,\alpha_2)}$ and $\chi^{(\alpha_2,\alpha_1)}$ will restrict to the same irreducible character of D_n for $\alpha_1 \neq \alpha_2$. The characters $\chi^{(\alpha,\alpha)}$ of B_n will restrict to a sum of two irreducible characters $\chi_+^{(\alpha,\alpha)}$ and $\chi_-^{(\alpha,\alpha)}$ of D_n. Note that in this case α is a partition of $n/2$ and again that this can only happen if n is even. We will in that case replace the second occurrence of α in the label by a sign to distinguish these two characters of D_n. The following function computes these character labels.

```
    wld.charparam[1]:= function(n)

        local alpha, labels;

        labels:= [];

        for alpha in PartitionTuples(n, 2) do
            if alpha[1] = alpha[2] then
                Add(labels, [alpha[1], '+']);
                Add(labels, [alpha[1], '-']);
            elif alpha[1] < alpha[2] then
                Add(labels, alpha);
            fi;
        od;

        return labels;

    end;
```

Last thing to do: the character values. But first we have to answer the question: What happens to the split characters on the split classes? As it turns out there is no such nice correspondence between the split classes and the split characters as in the case of alternating groups. But things are not very complicated here, too. We simply have to correct the restricted characters from B_n by character values of $S_{n/2}$.

Theorem 5.1 *Let $n > 0$ be an even integer and α, π be partitions of $n/2$. Then the character value $\chi_{\pm}^{(\alpha,\alpha)}$ on a class $(2\pi, [\,])_{\pm}$ is determined by the following formula.*

$$\chi_{\epsilon}^{(\alpha,\alpha)}((2\pi, [\,])_{\delta}) = \frac{1}{2}\chi^{(\alpha,\alpha)}(2\pi, [\,]) + \epsilon\delta 2^{l(\pi)-1}\chi^{\alpha}(\pi).$$

Proof. From [6] (2.45) we see that the characters are described as follows. We take $s = n/2$ copies of the trivial character 1 and also s copies of the sign character ϵ of S_2. Their product $1^s\epsilon^s$ describes a character of the elementary abelian normal subgroup S_2^n of B_n. Restrict this character to the intersection $\hat{2}$ of S_2^n and D_n which is a normal subgroup of D_n. Then extend the character to its inertia group T with inertia factor isomorphic to the wreath product $S_s \wr S_2$. This gives the character

$$\overline{(1^s\epsilon^s)\downarrow_{\hat{2}}}$$

of the inertia group. Furthermore we take the character $\chi^\alpha\chi^\alpha$ of $S_s \times S_s$ and extend it to its inertia group in $S_s \wr S_2$ which is $S_s \wr S_2$. This character is to be tensored with either the trivial or the sign character of S_2. Then the characters $\chi_{\pm}^{(\alpha,\alpha)}$ can be described as

$$\begin{aligned}
\chi_{+}^{(\alpha,\alpha)} &= \left(\overline{(1^s\epsilon^s)\downarrow_{\hat{2}}} \otimes \overline{\chi^\alpha\chi^\alpha} \otimes 1\right)^{D_n}, \\
\chi_{-}^{(\alpha,\alpha)} &= \left(\overline{(1^s\epsilon^s)\downarrow_{\hat{2}}} \otimes \overline{\chi^\alpha\chi^\alpha} \otimes \epsilon\right)^{D_n}.
\end{aligned}$$

We already know that the sum of these two characters yields character values of the character $\chi^{(\alpha,\alpha)}$ of B_n. We will now investigate the difference. Take a representative x of the class $(2\pi, [\,])$ of B_n for some partition π of $n/2$. We may assume that x is an element of the complement S_n and that it has a matrix representation of the form

$$\begin{bmatrix} 0 & X \\ I & 0 \end{bmatrix}$$

where X is a suitable permutation matrix with cycle structure π and I denotes the $n/2 \times n/2$ identity matrix. Then write the character values as a sum of character values of the inertia group T.

$$\begin{aligned}(\chi_+^{(\alpha,\alpha)} - \chi_-^{(\alpha,\alpha)})(x) &= \left(\overline{(1^s\epsilon^s)\restriction_{\hat{2}}} \otimes \overline{\chi^\alpha\chi^\alpha} \otimes (1-\epsilon)\right)^{D_n}(x)\\ &= |C_{D_n}(x)| \sum_{y\sim x} \frac{1}{|C_T(y)|}\overline{(1^s\epsilon^s)\restriction_{\hat{2}}}(y) \otimes \overline{\chi^\alpha\chi^\alpha}(y) \otimes (1-\epsilon)(y).\end{aligned}$$

Here the sum runs over those conjugacy class representatives y of T which are conjugate to x in D_n. The factor $\otimes(1-\epsilon)(y)$ will be 0 for those y with matrix representation

$$\begin{bmatrix} P_1 & 0 \\ 0 & P_2 \end{bmatrix}$$

for some signed permutation matrices P_1, P_2 such that the whole matrix has cycle structure 2π. But the only other representative of conjugacy classes of T that is conjugate to x in D_n is x itself. Hence there is only one summand, the factor $\overline{(1^s\epsilon^s)\restriction_{\hat{2}}}(x)$ will be 1 and with $\overline{\chi^\alpha\chi^\alpha}(x) = \chi^\alpha(\pi)$ we have for the difference of the character values

$$(\chi_+^{(\alpha,\alpha)} - \chi_-^{(\alpha,\alpha)})(x) = \frac{|C_{D_n}(x)|}{|C_T(x)|}\overline{(1^s\epsilon^s)\restriction_{\hat{2}}}(x) \otimes \overline{\chi^\alpha\chi^\alpha}(x) \otimes (1-\epsilon)(x) = \frac{|C_{D_n}(x)|}{|C_T(x)|} 2\chi^\alpha(\pi).$$

Finally observe that by (4.2)

$$\frac{|C_{D_n}(x)|}{|C_T(x)|} = \frac{|C_{S_n}(x)|}{|C_{S_s \wr S_2}(x)|} = \frac{|C_{S_n}(2\pi)|}{2|C_{S_s}(\pi)|} = \frac{\prod_i (2i)^{a_i(\pi)} a_i(\pi)!}{2\prod_i i^{a_i(\pi)} a_i(\pi)!} = \frac{1}{2}\prod_i 2^{a_i(\pi)} = 2^{l(\pi)-1}.$$

This completes the proof of Theorem 5.1. □

Thus the computation of the character values of D_n can be done with the functions that are already available. We start with a character value in B_n and for the split characters on the split classes this value will be corrected according to the above theorem.

```
wld.irreducibles[1][1]:= function(n, alpha, pi)

   local delta, val;

   if not IsList(alpha[2]) then
      delta:= [alpha[1], alpha[1]];
      if not IsList(pi[2]) then
         val:= wlb.irreducibles[1][1](n, delta, [pi[1], []])/2;
         if alpha[2] = pi[2] then
            val:= val + 2^(Length(pi[1])-1) *
                  sym.irreducibles[1][1](n/2, alpha[1], pi[1]/2);
         else
            val:= val - 2^(Length(pi[1])-1) *
                  sym.irreducibles[1][1](n/2, alpha[1], pi[1]/2);
         fi;
      else
         val:= wlb.irreducibles[1][1](n, delta, pi)/2;
      fi;
   else
      if not IsList(pi[2]) then
         val:= wlb.irreducibles[1][1](n, alpha, [pi[1], []]);
      else
         val:= wlb.irreducibles[1][1](n, alpha, pi);
      fi;
   fi;

   return val;

end;
```

As an example we can now compute and display the character table of the Weyl group of type D_4.

```
gap> d4:= CharTableSpecialized(wld, 4);
rec( name := "W(D4)", order := 192, centralizers :=
[ 192, 32, 192, 16, 8, 16, 32, 32, 16, 6, 6, 8, 8 ], orders :=
[ 1, 2, 2, 2, 4, 2, 2, 2, 4, 3, 6, 4, 4 ], powermap :=
[ , [ 1, 1, 1, 1, 2, 1, 1, 1, 3, 10, 10, 7, 8 ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 3, 12, 13 ] ], irreducibles :=
[ [ 3, -1, 3, -1, 1, -1, 3, -1, -1, 0, 0, -1, 1 ],
  [ 3, -1, 3, -1, 1, -1, -1, 3, -1, 0, 0, 1, -1 ],
  [ 4, 0, -4, -2, 0, 2, 0, 0, 0, 1, -1, 0, 0 ],
  [ 1, 1, 1, -1, -1, -1, 1, 1, 1, 1, 1, -1, -1 ],
  [ 6, -2, 6, 0, 0, 0, -2, -2, 2, 0, 0, 0, 0 ],
  [ 8, 0, -8, 0, 0, 0, 0, 0, 0, -1, 1, 0, 0 ],
  [ 3, 3, 3, -1, -1, -1, -1, -1, -1, 0, 0, 1, 1 ],
  [ 3, -1, 3, 1, -1, 1, 3, -1, -1, 0, 0, 1, -1 ],
  [ 3, -1, 3, 1, -1, 1, -1, 3, -1, 0, 0, -1, 1 ],
  [ 2, 2, 2, 0, 0, 0, 2, 2, 2, -1, -1, 0, 0 ],
  [ 4, 0, -4, 2, 0, -2, 0, 0, 0, 1, -1, 0, 0 ],
  [ 3, 3, 3, 1, 1, 1, -1, -1, -1, 0, 0, -1, -1 ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ], classparam :=
[ [ 1, [ [ 1, 1, 1, 1 ], [  ] ] ], [ 1, [ [ 1, 1 ], [ 1, 1 ] ] ],
  [ 1, [ [  ], [ 1, 1, 1, 1 ] ] ], [ 1, [ [ 2, 1, 1 ], [  ] ] ],
  [ 1, [ [ 1 ], [ 2, 1 ] ] ], [ 1, [ [ 2 ], [ 1, 1 ] ] ],
  [ 1, [ [ 2, 2 ], '+' ] ], [ 1, [ [ 2, 2 ], '-' ] ],
  [ 1, [ [  ], [ 2, 2 ] ] ], [ 1, [ [ 3, 1 ], [  ] ] ],
  [ 1, [ [  ], [ 3, 1 ] ] ], [ 1, [ [ 4 ], '+' ] ],
  [ 1, [ [ 4 ], '-' ] ] ], irredinfo := [ rec(
      charparam := [ 1, [ [ 1, 1 ], '+' ] ] ), rec(
      charparam := [ 1, [ [ 1, 1 ], '-' ] ] ), rec(
      charparam := [ 1, [ [ 1 ], [ 1, 1, 1 ] ] ] ), rec(
      charparam := [ 1, [ [  ], [ 1, 1, 1, 1 ] ] ] ), rec(
      charparam := [ 1, [ [ 1, 1 ], [ 2 ] ] ] ), rec(
      charparam := [ 1, [ [ 1 ], [ 2, 1 ] ] ] ), rec(
      charparam := [ 1, [ [  ], [ 2, 1, 1 ] ] ] ), rec(
      charparam := [ 1, [ [ 2 ], '+' ] ] ), rec(
      charparam := [ 1, [ [ 2 ], '-' ] ] ), rec(
      charparam := [ 1, [ [  ], [ 2, 2 ] ] ] ), rec(
      charparam := [ 1, [ [ 1 ], [ 3 ] ] ] ), rec(
      charparam := [ 1, [ [  ], [ 3, 1 ] ] ] ), rec(
      charparam := [ 1, [ [  ], [ 4 ] ] ] )
 ], text := "computed using generic character table for Weyl groups\
 of type D", classes := [ 1, 6, 1, 12, 24, 12, 6, 6, 12, 32, 32,
  24, 24 ], operations := CharTableOps )
gap> TestCharTable(d4);
true
```

```
gap> DisplayCharTable(d4);
W(D4)

     2  6  5  6  4  3  4  5  5  4  1  1  3  3
     3  1  .  1  .  .  .  .  .  .  1  1  .  .

       1a 2a 2b 2c 4a 2d 2e 2f 4b 3a 6a 4c 4d
    2P 1a 1a 1a 1a 2a 1a 1a 1a 2b 3a 3a 2e 2f
    3P 1a 2a 2b 2c 4a 2d 2e 2f 4b 1a 2b 4c 4d

X.1     3 -1  3 -1  1 -1  3 -1 -1  .  . -1  1
X.2     3 -1  3 -1  1 -1 -1  3 -1  .  .  1 -1
X.3     4  . -4 -2  .  2  .  .  .  1 -1  .  .
X.4     1  1  1 -1 -1 -1  1  1  1  1  1 -1 -1
X.5     6 -2  6  .  .  . -2 -2  2  .  .  .  .
X.6     8  . -8  .  .  .  .  .  . -1  1  .  .
X.7     3  3  3 -1 -1 -1 -1 -1 -1  .  .  1  1
X.8     3 -1  3  1 -1  1  3 -1 -1  .  .  1 -1
X.9     3 -1  3  1 -1  1 -1  3 -1  .  . -1  1
X.10    2  2  2  .  .  .  2  2  2 -1 -1  .  .
X.11    4  . -4  2  . -2  .  .  .  1 -1  .  .
X.12    3  3  3  1  1  1 -1 -1 -1  .  . -1 -1
X.13    1  1  1  1  1  1  1  1  1  1  1  1  1
```

The computation of the character table of the Weyl group of type D_6 (with 37 classes) needs half a minute.

```
gap> d6:= CharTableSpecialized(wld, 6);; time;
30850
```

The record `wld` is called `CharTableWeylD` in the GAP library. This record will be used by the function `CharTable` to compute a character table if it is called with first argument `"WeylD"`. Try to find a way how the fast computation of character values in B_n can be used in case D.

6 Cases E, F, G: Exceptional Weyl Groups.

There are five irreducible Weyl groups we have not covered yet. However, since this is a (rather) finite set we can treat them one by one. For each of these groups we will present a function which recovers the table from the GAP library. Moreover this function will for each irreducible character of the table add some information in the record field `irredinfo` which identifies the irreducible character with its label in Carter's book [2] (p. 411 ff.).

Such a label consists of two integers, the degree of the character and the first occurrence of the character as constituent in a symmetric power of the reflection character. The reflection character may be recognized from the character table as the character satisfying the following two conditions. Its degree d is the rank of the Weyl group (e.g. 6 in the case of E_6) and there is a class of involutions with character value $d-2$ (since the generating reflections of the Weyl group have eigenvalues 1 and one eigenvalue -1 in the reflection representation).

A quite efficient way to compute these labels is given by the concept of Molien series. The Molien series $M_{\psi,\chi}(z)$ in the ring of formal power series over the integers is defined to be the generating function of the multiplicities of χ as constituent of the d-th symmetric power $\psi^{[d]}$ of ψ and according to Molien can be computed as (see [7], p. 230)

$$M_{\psi,\chi}(z) = \frac{1}{|G|}\sum_{g\in G}\frac{\bar{\chi}(g)}{\det(I - zD(g))} = \frac{(-1)^{\psi(1)}}{|G|}\sum_{g\in G}\frac{\chi(g)\det D(g)}{\det(zI - D(g))}$$

where D is a matrix representation with character χ and if $\epsilon_1(D(g)),\ldots,\epsilon_{\psi(1)}(D(g))$ denote the eigenvalues of $D(g)$ then

$$\det(zI - D(g)) = \prod_{i=1}^{\psi(1)}(z - \epsilon_i(D(g))).$$

The eigenvalues corresponding to a character are determined by the GAP function `Eigenvalues`.

The Weyl group of type E_6 has order 51.840 and 25 conjugacy classes. It is isomorphic to an extension of the simple group $U_4(2)$ by a cyclic group of order two (see [3], p. 26 and cf. [1], p. 228). The character of the reflection representation is `X.4` and the generating reflections are contained in class `2c`. Its character table is restored by the following GAP function.

```
wle6:= function()

   local i, tbl, lbl;

   tbl:= CharTable("U4(2).2");
   tbl.name:= "W(E6)";

   lbl:= [0, 36, 9, 1, 25, 10, 5, 17, 4, 16, 2, 20, 6, 12, 3, 15, 8, 7,
     8, 5, 11, 4, 13, 6, 10];
   for i in [1..Length(lbl)] do
      tbl.irredinfo[i].label:= [tbl.irreducibles[i][1], lbl[i]];
   od;

   return tbl;

end;
```

The Weyl group of type E_7 has order 2.903.040 and 60 conjugacy classes. It is isomorphic to a direct product of the simple symplectic group $S_6(2)$ with a cyclic group of order two (see [3], p. 46 and cf. [1], p. 229). If the character table is constructed like below then the reflection character is `X.4` and the generating reflections are contained in class `2c`. Its character table can be constructed by the following function.

```
wle7:= function()

   local i, s, c, tbl, lbl;

   s:= CharTable("S6(2)");
   c:= CharTable("Cyclic", 2);
   tbl:= CharTableDirectProduct(s, c);
   tbl.name:= "W(E7)";
```

```
    lbl:= [0, 63, 46, 1, 28, 7, 6, 33, 36, 3, 2, 37, 22, 13, 4, 31, 30,
      3, 18, 9, 12, 15, 26, 5, 6, 21, 12, 15, 4, 25, 6, 21, 10, 17, 22,
      5, 20, 7, 6, 21, 10, 13, 16, 9, 18, 9, 8, 17, 16, 7, 14, 11, 14,
      9, 8, 15, 10, 13, 12, 11];
    for i in [1..Length(lbl)] do
       tbl.irredinfo[i].label:= [tbl.irreducibles[i][1], lbl[i]];
    od;

    return tbl;

end;
```

The Weyl group of type E_8 has order 696.729.600 and 112 conjugacy classes. It is isomorphic to $2.O_8^+(2).2$ (see [3], p. 85 and cf. [1], p. 228). The reflection character is not uniquely determined by the above conditions. But the character table has a table automorphism which allows to choose between the characters `X.68` and `X.69`. We take `X.68` as the reflection character. The generating reflections then are contained in class `2f`. The character table of this group is restored by

```
wle8:= function()

    local i, tbl, lbl;

    tbl:= CharTable("2.O8+(2).2");
    tbl.name:= "W(E8)";

    lbl:= [0, 120, 8, 68, 2, 74, 32, 8, 56, 4, 64, 24, 12, 36, 4, 52,
      20, 8, 44, 14, 38, 12, 36, 6, 46, 20, 16, 28, 6, 42, 20, 14, 26,
      22, 12, 32, 10, 34, 20, 8, 38, 20, 8, 32, 10, 34, 18, 16, 28, 18,
      10, 28, 16, 10, 30, 18, 14, 22, 18, 16, 22, 12, 26, 12, 24, 14,
      22, 1, 91, 19, 49, 3, 63, 7, 55, 25, 7, 43, 9, 39, 5, 47, 19, 13,
      31, 9, 39, 19, 13, 33, 11, 29, 7, 37, 17, 23, 13, 25, 19, 9, 31,
      13, 25, 17, 11, 27, 15, 21, 13, 23, 15, 21];
    for i in [1..Length(lbl)] do
       tbl.irredinfo[i].label:= [tbl.irreducibles[i][1], lbl[i]];
    od;

    return tbl;

end;
```

The Weyl group of type F_4 is a solvable group of order 1.152. It has 25 conjugacy classes and its character table is accessible via the name `"W(F4)"`. The reflection character can be chosen to be `X.17`. The generating reflections then lie in classes `2d` and `2f`.

```
wlf4:= function()

    local i, tbl, lbl;

    tbl:= CharTable("W(F4)");
    tbl.name:= "W(F4)";

    lbl:= [0, 24, [12,2], [12,1], [4,1], [16,2], [4,2], [16,1], 8,
           [6,1], [6,2], 2, [6,2], [6,1], 10, 4, 1, 13, [7,2], [7,1],
           [3,1], [9,2], [3,2], [9,1], 5];
    for i in [1..Length(lbl)] do
       tbl.irredinfo[i].label:= [tbl.irreducibles[i][1], lbl[i]];
```

```
      od;

      return tbl;

   end;
```

The following sequence of commands will sort this character table in exactly the same way as it is printed in [2] (p. 413).

```
   gap> f4:= wlf4();;
   gap> SortClassesCharTable(f4,
   > (4,6,10,8,5,9,7)(12,22,15,25,20,16)(13,23,14,24,19,18,17));;
   gap> SortCharactersCharTable(f4,
   > (2,4,3)(5,7)(6,8)(10,14,12)(11,15,13)(18,20,19)(21,23)(22,24);;
```

The Weyl group of type G_2 is isomorphic to the dihedral group D_{12} of order 12 with 6 conjugacy classes. Its character table is constructed from the generic character table for dihedral groups. The character of the reflection representation is `X.5` and and the generating reflections are contained in the classes `2b` and `2c`.

```
   wlg2:= function()

      local i, tbl, lbl;

      tbl:= CharTable("Dihedral", 12);
      tbl.name:= "W(G2)";

      lbl:= [0, 6, [3,1], [3,2], 1, 2];
      for i in [1..Length(lbl)] do
         tbl.irredinfo[i].label:= [tbl.irreducibles[i][1], lbl[i]];
      od;

      return tbl;

   end;
```

The following function will recover the list of labels from one of the above tables.

```
   LabelsWeyl:= function(tbl)

      local i, lbl;

      lbl:= [];
      for i in tbl.irredinfo do
         Add(lbl, i.label);
      od;

      return lbl;

   end;
```

Note that by checking these tables for the exceptional Weyl groups and the formulae in earlier sections the character values of all Weyl groups are rational.

References

[1] N. Bourbaki, *Groups et Algèbres de Lie, Chapitres 4, 5 et 6*, Masson, Paris 1981.

[2] R. W. Carter, *Finite Groups of Lie Type: Conjugacy Classes and Complex Characters*, Wiley–Interscience, New York 1985.

[3] J. H. Conway, R. T. Curtis, S. P. Norton, R. A. Parker, and R. A. Wilson, *Atlas of finite groups*, Clarendon Press, Oxford 1985.

[4] G. D. James and A. Kerber, *The Representation Theory of the Symmetric Group*, Encyclopedia of Math. 16, Addison–Wesley 1981.

[5] A. Kerber, *Representations of Permutation Groups I* LNM 240, Springer 1971.

[6] A. Kerber, *Representations of Permutation Groups II* LNM 495, Springer 1975.

[7] J. Neubüser, H. Pahlings, and W. Plesken, CAS; Design and Use of a System for the Handling of Characters of Finite Groups, in: *Computational Group Theory* (ed. M. Atkinson), Akad. Press 1984, 195 – 247.

[8] M. Schönert (ed.), GAP 3.1 *Manual*, Lehrstuhl D für Mathematik, RWTH Aachen 1992.