GAP code

This code was run in GAP version 4.8.7 on a PC (Microsoft Windows 10 Enterprise) with 3.2 GHz Intel quad core i5 and 16 GB of RAM. The generators created by "BminGenerators" are those found in Theorem 7.1. This code takes less than 15 minutes to execute. Attempting a computation in the rank-15 case yielded a memory overflow error.

```
#
# This function returns a list of n permutations which represent the simple
# reflections of the Weyl group of type B rank n acting on the weights of
# its minuscule representation. These generate the permutation representation
# of the Weyl group of type B rank n.
#
# s[i] corresponds to the simple reflection accross the hyperplane determined
# by the simple root alpha[i]. Since reflections are involutions (order 2),
# each permutation is the product of disjoint transpositions.
#
BminGenerators := function(n)
local s,i,tmp,j,k;

s := ListWithIdenticalEntries(n,[]);;
for i in [1..n-1] do
   tmp := ListWithIdenticalEntries(2^n,0);
   for j in [1..2^n] do
      tmp[j] := j;
   od;
   for j in [1..2^(n-1-i)] do
      for k in [1..2^(i-1)] do
         tmp[2^(i-1)+1+(j-1)*2^(i+1)+(k-1)] := 2^(i-1)+1+(j-1)*2^(i+1)+(k-1)+2^(i-1);
         tmp[2^(i-1)+1+(j-1)*2^(i+1)+(k-1)+2^(i-1)] := 2^(i-1)+1+(j-1)*2^(i+1)+(k-1);
      od;
   od;
   s[n-i] := PermList(tmp);
od;;

tmp := ListWithIdenticalEntries(2^n,0);;
for j in [1..2^(n-1)] do
   tmp[2*j-1] := 2*j;
   tmp[2*j] := 2*j-1;
od;;
s[n] := PermList(tmp);;

return s;

end;;
```

```
#
# Given a permutation s and rank n, this function determines how many of each
# type of cycle appears in s. Since we want to keep track of 1-cycles (which
# are normally suppressed), we need the rank to find out how many integers in
# the list 1..2^n are unmoved (i.e. the number of 1-cycles).
#
# This function returns a list of pairs of the form "[k,m]" which indicates
# that the permutation has m k-cycles.
#
# For example: s=(1,2,3)(4,5,6) and n=4 means s=(1,2,3)(4,5,6)(7)(8)...(16)
# so the function returns [[1,16],[3,2]] (16 1-cycles and 2 3-cycles).
#
CycleType := function(s,n)
local tmp,lst,i,z;

# CycleStructurePerm returns a list of the number of cycles of each type
# starting with transpositions.
tmp := CycleStructurePerm(s);

# lst = [0,tmp]
# The "0" will be replaced by the number of 1-cycles.
lst := [0];;
Append(lst,tmp);;

# This replaces empty spots in lst with 0's.
z := Zero([1..Length(lst)]);
lst := lst+z;

tmp := 0;;
for i in [1..Length(lst)] do
   # i*lst[i] is the number of integers moved by the i-cycles.
   tmp := tmp+i*lst[i];
od;
# tmp is the total number of integers in 1..2^n moved by non-trivial cycles,
# so 2^n-tmp is the number of 1-cycles (trivial cycles).
lst[1] := 2^n-tmp;;

# This converts our list of numbers of k-cycles to a more convenient format.
# If list[k]=m > 0 then we add "[k,m]" to our list signifying that there
# are a total of m k-cycles. So [3,5,0,7] turns into [[1,3],[2,5],[4,7]].
tmp := [];
for i in [1..Length(lst)] do
   if not lst[i] = 0 then
      Append(tmp,[[i,lst[i]]]);
   fi;
```

```
od;;
lst := tmp;

return lst;

end;;


#
# This function returns the distinct cycle types that appear in the minuscule
# permutation representation of the Weyl group of type B rank n.
#
# For example: When n=2, we get [[[1,2],[2,1]], [[1,4]], [[2,2]], [[4,1]]].
# This means that the permutation representation contains permutations of the
# form... (A) 2 1-cycles and a transposition, (B) 4 1-cycles (the identity),
# (C) 2 tranpositions, and (D) 1 4-cycle.
#
BminCycleTypes := function(n)
local ccl,csl,cycTypes,k;

# This the a complete list of the conjugacy classes of our perm. rep.
ccl := ConjugacyClasses(Group(BminGenerators(n)));;

# csl is a list of representatives -- one from each conjugacy class.
csl := List(ccl, c -> Representative(c));;

# We compute the cycle type of each representative in csl and add it to our
# list of cycle types: cycTypes.
cycTypes := [];;
for k in [1..Length(csl)] do
   Append(cycTypes,[CycleType(csl[k],n)]);
od;

# Elements of two distinct conjugacy classes can share the same cycle type.
# Thus we apply SSortedList to remove redundancies in our list.
return SSortedList(cycTypes);

end;;


#
# We know that the Weyl group acts transitively on the set of weights of a
# minuscule representation. So there are no non-empty proper subsets of
# weights left invariant under the group's action. In some cases, this is
# visible from the cycle structures (of the Weyl group elements realized
# as permutations) alone.
```

```
#
# This function returns a list of sizes of invariant subsets of weights
# allowed by the cycle structures of the perm. rep. of the Weyl group of
# type B rank n acting on the weights of its minuscule representation.
#
BminInvSubspDim := function(n)
local cycTypes,subsp,m,elt,myList,i,indicesOfInterest,j,k,tmp,subspTMP;

# Get the cycle types for the perm. rep.
cycTypes := BminCycleTypes(n);

# no elements (yet) ==> all subset sizes are allowed.
subsp := [0..2^n];;

for m in [1..Length(cycTypes)] do

   # grab a cycle type.
   elt := cycTypes[m];

   # myList is a list of 2^n+1 copies of "false". myList[i+1] corresponds
   # to an allowed invariant subset of size i.
   myList := ListWithIdenticalEntries(2^n+1,false);;

   # the empty set is always allowed.
   myList[1] := true;;

   for i in [1..Length(elt)] do   # i-th type of cycle in elt

      # look through myList and grab only the indices y for which myList[y] is true.
      indicesOfInterest := Filtered([1..Length(myList)], y -> myList[y]);

      for j in indicesOfInterest do   # all j's where myList[j]=true
         # If elt[i]=[x,y], then elt has y x-cycles, so k goes from 1 to y which
         # happens to be the number of x-cycles.
         for k in [1..elt[i][2]] do
            # Suppose elt[i]=[x,y]. We know myList[j]=true (an invariant
            # subset of size j is allowed by elt). If we let in anything from
            # an x-cycle, we must allow all x elements from that cycle. So
            # if j is allowed, then so is j+x (but nothing between j and j+x).
            # Looping through all y x-cycles, we get j,j+x,j+2x,...,j+yx are all
            # allowed.
            myList[j+k*elt[i][1]] := true;
         od;
      od;
   od;
   # tmp is a list of indices corresponding to invariant subset sizes allowed
   # by the permutation elt.
   tmp := Filtered([1..Length(myList)], y -> myList[y]);;

   # since the y-th element corresponded to a set of size y-1 we need to decrease
   # everything in tmp by 1.
   tmp := List(tmp, p -> p-1);

   # subspTMP is the list of common subset sizes allowed by previous elements.
   subspTMP := subsp;

   subsp := [];
   for i in tmp do   # size "i" is allowed by elt (it appears in tmp)

     # If size "i" was allowed by all previous elements, we should add it
     # to our list of allowed sizes.
     if i in subspTMP then
        Add(subsp,i);
     fi;
   od;
od;;

return subsp;  # The listed sizes were allowed by all of the cycle types in cycTypes.

end;;


#
# This returns in the order of the Weyl group of type B rank n.
#
BWeylSize := function(n)
   return(Size(Group(BminGenerators(n))));
end;;

#
# This returns a permutation representing the Coxeter element of the Weyl group
# of type B rank n. This is just the product of all of the simple reflections:
# s[1]s[2]...s[n].
#
BminCoxeter := function(n)
local s,coxeter,i;

s := BminGenerators(n);

coxeter := (1);;
for i in [1..n] do
```

```
   coxeter := coxeter*s[i];
od;;

return coxeter;

end;;

#
# Let’s see what dimensions are allowed for the first 14 ranks...
#
for n in [1..14] do
  Print("B",n,": ",BminInvSubspDim(n),"\n");
od;
```