Prolog Programming Contest: 17 August, Seattle

Prologue

There are 7 problems. The name of the file that contains your submitted solution must be the same as given in the header of the problem - this file must be readable for the organisers. So, for the first problem, you make a file named `cake.pl`.
The *input* to your program is always in the form of one or more arguments to the predicate you must write.
Each correct submission (at most one for each problem) earns you one point.
Efficiency of your programs is not important (except in the Berge sorting problem), but if your program fails to finish in a reasonable time, it will be considered incorrect. You can earn a bonus for problem 2 (Berge sorting): in case of a tie otherwise, the speed of your solution will break it.
Do not start predicate names with iclp06 or test - never use the dynamic database or other equivalent global stuff built-in predicates !

1 The Birthday Cake *(cake.pl)*

We celebrate an important birthday this year and that's why you need to write a predicate cake/1 which takes as input a number, and outputs a birthday cake of that size. The specification of how the birthday cake looks is in the following pictures:

```
?- cake(2).

 /\  /\  /\  /\  /\
 \/  \/  \/  \/  \/
 ||  ||  ||  ||  ||
 ||  ||  ||  ||  ||
oooooooooooooooooooo
o/\oo/\oo/\oo/\oo/\o
o\/oo\/oo\/oo\/oo\/o
oooooooooooooooooooo
```

Size 2 is the smallest cake we'll ask you to draw.

```
?- cake(3).

  /\    /\    /\    /\    /\
  \/    \/    \/    \/    \/
  ||    ||    ||    ||    ||
  ||    ||    ||    ||    ||
  ||    ||    ||    ||    ||
  ||    ||    ||    ||    ||
oooooooooooooooooooooooooooooo
oo/\oooo/\oooo/\oooo/\oooo/\oo
o/xx\oo/xx\oo/xx\oo/xx\oo/xx\o
o\xx/oo\xx/oo\xx/oo\xx/oo\xx/o
oo\/oooo\/oooo\/oooo\/oooo\/oo
oooooooooooooooooooooooooooooo
```

As you can see, a birthday cake has 5 candles (one for each decade) and their size is directly related to the input to cake/1. So is the number of layers of the cake. We wish those concerned a happy birthday !
Just so that you know you are on the right track, here is another example cake:

```
?- cake(4).

   /\      /\      /\      /\      /\
   \/      \/      \/      \/      \/
   ||      ||      ||      ||      ||
   ||      ||      ||      ||      ||
   ||      ||      ||      ||      ||
   ||      ||      ||      ||      ||
   ||      ||      ||      ||      ||
   ||      ||      ||      ||      ||
oooooooooooooooooooooooooooooooooooooooo
ooo/\oooooo/\oooooo/\oooooo/\oooooo/\ooo
oo/xx\oooo/xx\oooo/xx\oooo/xx\oooo/xx\oo
o/xxxx\oo/xxxx\oo/xxxx\oo/xxxx\oo/xxxx\o
o\xxxx/oo\xxxx/oo\xxxx/oo\xxxx/oo\xxxx/o
oo\xx/oooo\xx/oooo\xx/oooo\xx/oooo\xx/oo
ooo\/oooooo\/oooooo\/oooooo\/oooooo\/ooo
oooooooooooooooooooooooooooooooooooooooo
```

2 Berge Sorting *(berge.pl)*

N pegs are in adjacent peg holes. If N is even, there are as many black pegs as white pegs. If N is odd, there is one more white peg. The pegs are placed so that the colours alternate and the leftmost peg is white. For $N = 5$, Figure 1 should make sense now.

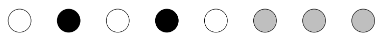

Figure 1: Alternating white-black pegs and some empty ones

The peg holes actually extend to the right as far as you want: in picture, such empty peg holes are shaded. You can move pegs as follows: pick up to adjacent pegs, and move them (without altering their order) to two adjacent empty peg holes. The goal is to *Berge sort* the pegs, meaning that through a series of moves, the pegs are in adjacent holes, and all white pegs are to the left of all the black pegs (or the other way around). Figure 2 should give you an idea of how this can be done:

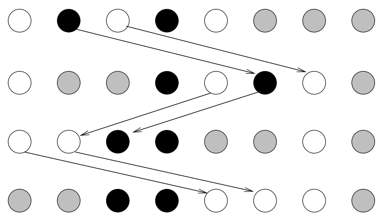

Figure 2: Moves that sort the pegs

Note that in this figure, sorting is done in 3 steps: that's optimal. In general, one needs at most $(N + 1)//2$ steps and there is also a known upper-bound on the number of extra empty pegs you need to achieve this - all this is true for $N > 4$ - but the latter should not worry you: we'll provide you with enough of these empty peg holes. This upper bound on the minimal number of steps required to Berge sort is quite amazing given the fact that half of the black pegs are in the wrong position, and so are half of the white pegs, and you can move only two pegs in one step !
You represent a sequence of peg holes with pegs, as for example:

```
[white,empty,empty,black,white,black,white,empty]
```

That is the representation of the second row in Figure 2. You write a predicate bsort/2 which takes as input an initial configuration, like

```
?- bsort([white,black,white,black,white,empty,empty,empty],L).
```

(note that it has enough empty peg holes - actually one more than you really need). bsort/2 unifies its second argument with a list of all the configurations (including the initial one) you go through while Berge-sorting the initial one - and up to the sorted one. For the example above, this would be:

```
L =
      [[white,black,white,black,white,empty,empty,empty]
       [white,empty,empty,black,white,black,white,empty]
       [white,white,black,black,empty,empty,white,empty]
       [empty,empty,black,black,white,white,white,empty]]
```

Your solution must have the minimal number of steps. For most N, there is not a unique optimal solution: just give us one.

3 Hereditary Base Notation ***hbn.pl***

You can denote any number in base 3 notation, e.g. $16_{10} = 121_3$ and we can alternatively write this as $16 = 3^2 + 2 * 3^1 + 1$ (where all numbers are written in their base 10 notation). You might quibble with us about whether also $16 = 1 * 3^2 + 2 * 3^1 + 1 * 3^0$ is ok, and we agree: that's ok too, as well as some other forms. The essence is: it is a sum of (different) powers of 3 and factors in front are smaller than 3. Now take a larger number, e.g. 2187, then you get $2187 = 3^7$ and now we have used a number larger than 3, and we don't want that: we replace that 7 by a sum of powers of 3 - you know how that works now. So since $7 = 1 + 2 * 3^1$ we get: $2187 = 3^{1+2*3^1}$. And $6561 = 3^{2*3^1+2}$. So, once you have done that recursively for a given number w.r.t. some base, you get its hereditary base notation. We want you to write a predicate hbn/3 which takes as input a number and a base, and unifies its third argument with the number's hereditary base notation w.r.t. the given base. The Prolog representation of a hereditary base notation is as follows:

- a sum of things is represented by a list of these things - since addition is commutative, the order does not matter; so (a+b+c) is represented by [a,b,c] or [b,a,c] ...
- the arithmetic expression X^Y is represented by the Prolog term `A ^ B`; in a hbn, the only exponentiation happens to the base, so, A is always equal to the given base; B is the hbn of Y
- the arithmetic expression $X * Y$ is represented by the Prolog term `X * A`; since X is always smaller than the base, X is just this number; and A is the hbn of Y (which is always a power of the base)
- a single number smaller than the base is represented by itself or by a list containing only the number

These rules are a nondeterministic specification of how an hbn should look as a Prolog term. So, some examples might be useful:

```
?- hbn(10,4,L).
L = [2,2 * 4 ^ [1]]

?- hbn(7625597484987,3,L).
L = [1 * 3 ^ [1 * 3 ^ [1 * 3 ^ [1]]]]
```

The last answer may also be written as

```
 L = [3 ^ [3 ^ [3]]]
```

and some other forms would also do.

4 The Goodstein sequence *goodstein.pl*

You probably should first tackle problem 3, but read on to know why ...
Once you have written the hbn predicate from problem 3, you can implement the Goodstein sequence generator. You are given a number and a base - e.g. 4 and 2. Start with computing its hbn - this gives you (in normal representation - not the one of the previous problem) 2^2. Add 1 to the base - you get the new base 3. Change in the current hbn every occurrence of the old base to the new base. You get 3^3 (which is equal to 27). Now subtract 1 and make the hbn in the **new** base - giving you $26 = 2 * 3^2 + 2 * 3^1 + 2$. Repeat ...
Let's do this a couple of times more and in a table

value	base	hbn	new base	changed hbn	1 subtracted	new value
4	2	2^2	3	3^3	$2 * 3^2 + 2 * 3^1 + 2$	26
26	3	$2 * 3^2 + 2 * 3^1 + 2$	4	$2 * 4^2 + 2 * 4^1 + 2$	$2 * 4^2 + 2 * 4^1 + 1$	41
41	4	$2 * 4^2 + 2 * 4^1 + 1$	5	$2 * 5^2 + 2 * 5^1 + 1$	$2 * 5^2 + 2 * 5^1$	60
60	5	$2 * 5^2 + 2 * 5^1$	6	$2 * 6^2 + 2 * 6^1$	...	...

The numbers in the value column form are the first 4 of the Goodstein sequence with seed 4 and base 2: G(4,2). We want you to write a predicate goodstein/4 that takes a seed S, a base B and a number N, and produces in a list the first N elements of G(S,B). So for instance:

```
?- goodstein(4,2,4,L).
L = [4,26,41,60]

?- good(10,2,9,L).

L = [10,83,1025,15625,279935,4215754,84073323,1937434592,50000555551]
```

You might think that Goodstein sequences have elements that grow forever, but Goodstein proved (in 1944 !) that every sequence (whichever seed and base) eventually becomes zero: you can always stop there. We'll test your program only with positive S, B and N.

5 Von Koch's snowflake *(snow.pl)*

You certainly know the von Koch snowflake, but just to be sure, this is about a sequence of pictures which starts with a triangle and where each next picture can be seen as a variation on the previous one ... too cumbersome to explain this in detail. Look at Figure 3 and you see the initial triangle (a level 1 snowflake), and the next three levels of von Koch snowflakes.

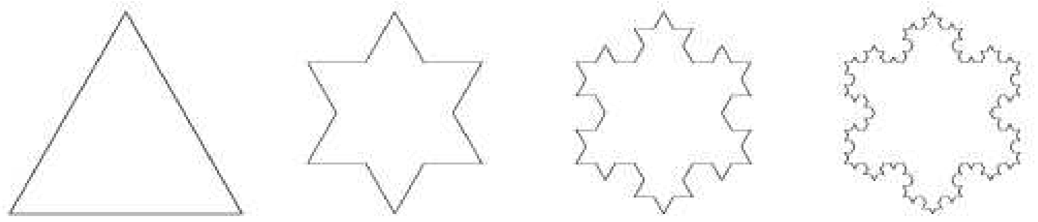

Figure 3: 4 levels of von Koch snowflakes

Instructions for drawing a level 1 snowflake can be simply represented by $[60, -60, 180]$. These are just the angles (with a horizontal line) of line segments of the triangle, as you go along drawing the triangle starting at the leftmost corner and going clock wise. It is implicit that all line segments are of equal length. Your predicate snowflake/2 should take as first argument an N, the level of a snowflake, and unify its argument with the instructions for drawing an N level Koch snowflake in the prescribed manner. So for instance

```
?- snowflake(1,L).
        L = [60,-60,180]
?- snowflake(2,L).
        L = [60,120,0,60,-60,0,-120,-60,180,-120,120,180]
```

Angles should always be in the range]-180,180].

6 Ukodus *(ukodus.pl)*

You need to write a predicate ukodus/2 which solves the following puzzle: given a multi-set of numbers, (1) put them in a grid such that no column neither row contains twice the same number, and (2) such that the space occupied by the numbers is minimal. The first requirement is easy to fulfil: suppose the numbers $7, 7, 8$ are given, then you can put them in a grid as in the left of Figure 4. However, that configuration occupies an enclosing rectangle (in this case it is a square) of size 9. If you put the numbers as in the right of Figure 4, they occupy only 4. That is also a minimum.

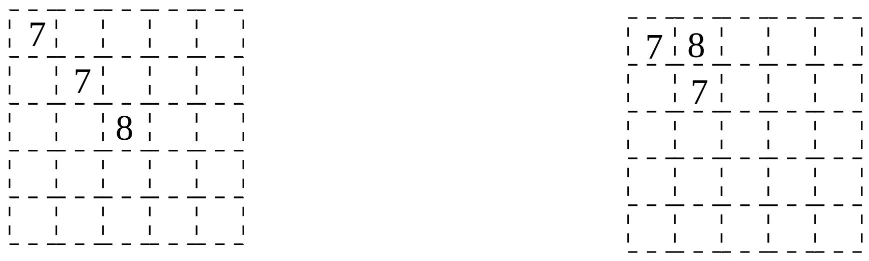

Figure 4: Putting [7,7,8] in a grid

The multi-set is given as a list of numbers, in the above example it could be any permutation of $[1, 1, 2]$. Your predicate ukodus/2 should unify its second argument with a configuration, whose representation is a list with elements of the form square(X,Y,N) where (X,Y) are coordinates in the grid and N is the number you have put there. For instance, the left configuration in Figure 4 is represented by `[square(1,1,7), square(2,2,7), square(3,3,8)]` (or a permutation thereof). You are free to choose your origin and the direction in which your coordinates increase, so also `[square(0,0,7), square(-1,-1,2,7), square(-2,-2,8)]` would be ok. Here is one more example: input is [1,2,3,1,2,3,1,2,3,1,2,3,4]. Some configurations satisfying (1) are shown in Figure 5. The one on the right is an optimal one. You can also see that you don't have to make the filled out squares a connected set.

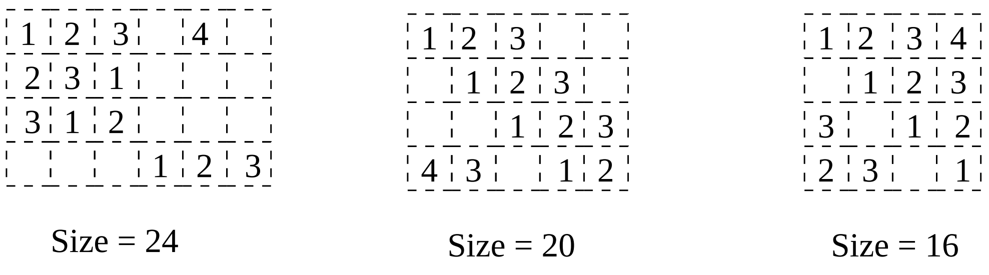

Figure 5: Putting [1,2,3,1,2,3,1,2,3,1,2,3,4] in a grid

7 Soccer *(soccer.pl)*

Somewhere in the middle of the soccer world championship, there are 2^N teams left, and the direct elimination phase starts. We'll focus on this phase. When the champion is known, a table is made which contains for each team its name, the number of matches the team has played, the number of goals the team made and the number of goals it took. Here is an example:

```
monaco        2 10 2
andorra       2  6 4
liechtenstein 1  0 7
sanmarino     1  1 4
```

Such a table might be the result of the matches below:

```
monaco-andorra        3-2
monaco-liechtenstein 7-0
andorra-sanmarino    4-1
```

The problem you must solve is indeed: given a table, reconstruct the matches. It is clear that this is not always uniquely possible, but any solution will do.
The predicate that you must write is named soccer/2. It will be called with as first argument a Prolog term representation of the table, and with free second argument, as in

```
?- soccer([result(monaco,2,10,2),
           result(andorra,2,6,4),
           result(liechtenstein,1,0,7),
           result(sanmarino ,1,1,4)],
           Matches).
```

and it should unify the second argument with a representation of the matches, in this case it could be

```
Matches = [monaco-andorra = 3-2,
           monaco-liechtenstein = 7-0,
           andorra-sanmarino  = 4-1]
```

The order in the input is immaterial, and so is the order in the output. Also, there is no need to put the winner first in the representation of a game, so monaco-andorra = 3-2 is the same as andorra-monaco = 2-3.
Of course, games never end in a draw.