

Prolog Programming Contest: 17 December 2008, Udine

There are 5 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 `chocolate.pl`.

The *input* to your program is in the form of one or more arguments to the predicate you must write, or as some given facts: do not put such given facts in any submission.

Each correct submission (at most one for each problem) earns you one point. Incorrect submissions cost nothing, except that you might be mentioned in the award speech.

Efficiency of your programs is usually not important, but if your program fails to finish in a reasonable time, it will be considered incorrect. However, the efficiency of your submission to problem *chain* will decide in case of a draw.

Do not start predicate names with `iclp08` or `test` - do not use the dynamic database or other similar global stuff built-in predicates!

1 Chocolate (*chocolate.pl*)

Below you see a chocolat bar of size 4x3: your general `chocolate/2` predicate should be able to generate it on the screen by the goal `?- chocolate(4,3)`.

```

  ---  ---  ---  ---
.'\_\_\'\_\_\'\_\_\'\_\_\\
|\V  __V\  __V\  __V\  __\
  \'\_\_\'\_\_\'\_\_\'\_\_\\
    \V  __V\  __V\  __V\  __\
      \'\_\_\'\_\_\'\_\_\'\_\_\\
        \V  \V  \V  \V  \
          \|\_-----|
```

Of course, other reasonable input parameters must be dealt with sensibly ! Just as another example: `?- chocolate(10,5)`.

```

  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---
.'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\\
|\V  __V\  __V\  __V\  __V\  __V\  __V\  __V\  __V\  __\
  \'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\\
    \V  __V\  __V\  __V\  __V\  __V\  __V\  __V\  __V\  __\
      \'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\\
        \V  __V\  __V\  __V\  __V\  __V\  __V\  __V\  __V\  __\
          \'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\'\_\_\\
            \V  \V  \V  \V  \V  \V  \V  \V  \V  \V  \
              \|\_-----|
```

2 Pixels (*pixels.pl*)

Once more, the result of this problem is an ascii picture on the screen. But this time, there is a real computation behind it ... You get a large square that is divided in N by M little squares. We refer to the little squares as pixels. Initially, the pixels are blanc, or contain a number larger than (or equal to) 1. The picture you show on the screen must have every pixel lit up that is on a valid path between two pixels (including them) with the same number in it.

A valid path between two pixels containing number n is a path with length equal to n . Paths are formed by going from a pixel to a neighbouring pixel: the neighbours are the left, right, upper and lower pixels. Two paths cannot have a pixel in common.

Maybe time for some concrete examples. The following term

```
square(pixels(3,_,2),
       pixels(_,_,2),
       pixels(3,_,_))
```

represents a 3 by 3 square, with blanc pixels represented as a void variable; there are 2 pixels with a 3, and two with a 2. The only picture that can be produced by the query ? - *pix(< the above term >)*. is:

```
* *
 *
 *
```

You see that lit up pixels are shown as stars. The input

```
square(pixels(5,_,_,5,_,_,4,_,_,4,_,_,4,_,_,_,_,5,_,_,_,_,4,3,_,3),
       pixels(_,_,_,_,_,3,_,_,_,_,_,_,_,_,_,_,_,_,_,3),
       pixels(_,2,2,_,_,_,3,_,3,_,_,_,_,_,_,_,_,_,_,_),
       pixels(_,_,_,_,_,3,_,_,_,_,4,_,_,_,_,_,_,_,4,_,_,3),
       pixels(5,_,_,5,_,_,4,_,_,4,_,_,4,_,_,4,_,_,5,3,_,3,_,_,4,_,_,4))
```

results in

```
* * **** * * ****
* * * * * * * *
**** **** * * * *
* * * * * * * *
* * **** **** **** ****
```

Note that input like

```
square(pixels(3,_,3),
       pixels(_,_,_),
       pixels(3,_,_))
```

cannot result in a picture. That's why the input always contains an even number of pixels with a particular number larger than 1.

3 Largests Chain of Permutations (*chain.pl*)

A permutation $[b,c,d,e,a]$ of a list $[a,b,c,d,e]$ can be specified by giving the position list P $[2,3,4,5,1]$ which says: the permutation contains in its first position the second item of the original list, in second position the 3th item, ... and in the fifth place the first item. One can apply the permutation many times starting from the initial list, and after a number T times doing that, one gets the initial list again: in this case T equals 5. You must write a predicate *large/3* that for a given number N (which was 5 in the example), computes a position list P that maximizes T . For example:

```
?- large(5,P,T).
T = 6
P = [3,4,5,2,1]
```

would be correct. But there are other positions lists P that are equally correct.

4 Largest Flip Sum (*flip.pl*)

You get some decimal numbers, all of the same length and written under each other - some might begin with one or more zeroes, as in left part of the figure below:

12345	17654
23456	73456
00934	90934
76543	26543
30303	60303

You can easily take the sum of those numbers: 143581. Two transformations of the numbers are allowed: you can *flip* all digits in a column, and you can flip all digits in a row. Flipping a digit n means replacing it by $9 - n$. So you should be able to check that flipping the first row and the first column of the left numbers results in the right numbers above.

You can also easily check that this results in a different sum of the resulting numbers: 268890. Clearly, there is a maximal sum S_{max} that can be obtained if you are allowed any number of transformations, but of course, since flipping the same row or column twice is a noop, you can flip each row or column only once. Finally, to obtain S_{max} , there is a minimal and a maximal number of flips needed: we want you to compute them both. To be more precise: look at the following query and its answer

```
?- flip([[1,2,3,4,5],[2,3,4,5,6],[0,0,9,3,4],[7,6,5,4,3],[3,0,3,0,3]],
        MaxSum, MinFlips,MaxFlips).
MaxSum = 409604
MinFlips = 4
MaxFlips = 6
```

Please write *flip/4*.

5 Shortest Dice Throw Distance (*dice.pl*)

You know these games in which you have to make a plan to achieve a particular goal, and at each turn, you must throw a dice and then you can only execute a part of the plan, according to your dice throw. It is very annoying, because your throws can really ruin your plan. That's why in this game, you get a sequence of dice throws in advance. And now the game itself ... every game is in some way equivalent to a path finding problem in a graph, so we haven't bothered with making it more concrete than this: the graph is directed, you start at node a and you need to go to node z and on your n^{th} turn, you must follow as many arcs as the n^{th} element of the given die throw sequence tells you. You may during that turn run in circles, even go back and forth between two nodes (if the arcs permit that) but you must make this exact number of transitions from one node to another. Your task is to get to z in the minimal number of turns. You need to know two more things about this game: you get a finite sequence of dice throws, but you must consider it as a cyclic sequence, i.e., if you need more turns than the length of the sequence, you can restart the sequence. Secondly, it is possible that the sequence (even if repeated many times) cannot lead you to z . In that case, your program should finitely fail (see an example later). The graph is given as a list of the form $[\text{arc}(a,b), \text{arc}(b,z)]$ (meaning the graph has one arc from a to b and one from b to z). The sequence is given as a finite list of strictly positive integers. You write a predicate `dice/3` with first argument the graph, second argument the sequence, and the third argument must be unified with the minimal number of turns needed to go from a to z . Some examples:

```
?- dice([arc(a,b), arc(b,a), arc(b,z), arc(z,b)], [3,5], N).  
N = 2
```

```
?- dice([arc(a,b), arc(b,z)], [4,2,6], N).  
No
```

```
?- dice([arc(a,b), arc(b,c), arc(c,z)], [1], N).  
N = 3
```